Postel

SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM

December 1974 Revision

Vinton Cerf
Yogen Dalal
Carl Sunshine

INWG 72

ARPA NWG

RFC 675

TABLE OF CONTENTS

1. Introduction page 3
2. The TCP Interface to the User page 5
   2.1 The TCP as a post office page 5
   2.2 Sockets and Addressing page 5
   2.3 TCP user calls page 7
      2.3.1 A Note on style page 7
      2.3.2 Open connection page 8
      2.3.3 Send letter page 9
      2.3.4 Receive letter page 10
      2.3.5 Close connection page 12
      2.3.6 Interrupt page 12
      2.3.7 Status page 13
   2.4 TCP to user messages page 13
      2.4.1 Type codes page 13
      2.4.2 Message format page 13
      2.4.3 Event codes page 14
3. Higher level protocols page 16
   3.1 Introduction page 16
   3.2 Well known sockets page 16
   3.3 Reconnection protocol page 17
4. TCP implementation page 18
   4.1 Introduction page 18
   4.2 TCP data structures page 19
      4.2.1 Internetwork packet format page 19
      4.2.2 Transmission control block page 22
   4.3 Connection management page 24
      4.3.1 Initial sequence number selection page 24
      4.3.2 Establishing a connection page 25
      4.3.3 Half-open connections page 27
      4.3.4 Resynchronizing a connection page 28
      4.3.5 Closing a connection page 28
      4.3.6 Connection states page 30
   4.4 Structure of the TCP page 32
      4.4.1 Introduction page 32
      4.4.2 Input packet handler page 35
      4.4.3 Reassembler page 36
      4.4.4 Packetizer page 37
      4.4.5 Output packet handler page 38
      4.4.6 User call processing page 39
      4.4.7 Network control processing page 41
      4.4.8 TCP error handling page 41
   4.5 Buffer and window allocation page 42
      4.5.1 Introduction page 42
      4.5.2 The send side page 42
      4.5.3 The receive side page 43
5. Network measurement plans for the TCP page 45

5.1 User level diagnostics page 45
5.2 Single connection measurements page 45
5.3 Multiconnection measurements page 47
5.4 Measurement implementation philosophy page 47
6. Schedule of implementation page 47
7. References page 47

1. INTRODUCTION

This document describes the functions to be performed by the internetwork Transmission Control Program [TCP] and its interface to programs or users that require its services. Several basic assumptions are made about process to process communication and these are listed here without further justification. The interested reader is referred to [CERA74, TOML74, BELS74, DALA74, SUNS74] for further discussion.

The authors would like to acknowledge the contributions of R. Tomlinson (three way handshake and Initial Sequence Number Selection), D. Belsnes, J. Burchfiel, M. Galland, R. Kahn, D. Lloyd, W. Plummer, and J. Postel all of whose good ideas and counsel have had a beneficial effect (we hope) on this protocol design. In the early phases of the design work, R. Metcalfe, A. McKenzie, H. Zimmerman, G. LeLann, and M. Elie were most helpful in explicating the various issues to be resolved. Of course, we remain responsible for the remaining errors and misstatements which no doubt lurk in the nooks and crannies of the text.

Processes are viewed as the active elements of all HOST computers in a network. Even terminals and files or other I/O media are viewed as communicating through the use of processes. Thus, all network communication is viewed as inter-process communication.

Since a process may need to distinguish among several communication streams between itself and another process [or processes], we imagine that each process may have a number of PORTs through which it communicates with the ports of other processes.

Since port names are selected independently by each operating system, TCP, or user, they may not be unique. To provide for unique names at each TCP, we concatenate a NETWORK identifier, and a TCP identifier with a port name to create a SOCKET name which will be unique throughout all networks connected together.

A pair of sockets form a CONNECTION which can be used to carry data in either direction [i.e. full duplex]. The connection is uniquely identified by the <local socket, foreign socket> address pair, and the same local socket can participate in multiple connections to different foreign sockets [see Section 2.2].

Processes exchange finite length LETTERS as a way of communicating; thus, letter boundaries are significant. However, the length of a letter may be such that it must be broken into FRAGMENTS before it can be transmitted to its destination. We assume that the fragments will normally be reassembled into a letter before being passed to the receiving process. Throughout this document, it is legitimate to

assume that a fragment contains all or a part of a letter, but that a fragment never contains parts of more than one letter.

We specifically assume that fragments are transmitted from Host to Host through means of a PACKET SWITCHING NETWORK [PSN] [ROWE70,POUZ73]. This assumption is probably unnecessary, since a circuit switched network could also be used, but for concreteness, we explicitly assume that the hosts are connected to one or more PACKET SWITCHES [PS] of a PSN [HEKA70, POUZ74, SCWI71].

Processes make use of the TCP by handing it letters. The TCP breaks these into fragments, if necessary, and then embeds each fragment in an INTERNETWORK PACKET. Each internetwork packet is in turn embedded in a LOCAL PACKET suitable for transmission from the host to one of its serving PS. The packet switches may perform further formatting or other operations to achieve the delivery of the local packet to the destination Host.

The term LOCAL PACKET is used generically here to mean the formatted bit string exchanged between a host and a packet switch. The format of bit strings exchanged between the packet switches in a PSN will generally not be of concern to us. If an internetwork packet is destined for a TCP in a foreign PSN, the packet is routed to a GATEWAY which connects the origin PSN with an intermediate or the destination PSN. Routing of internetwork packets to the GATEWAY may be the responsibility of the source TCP or the local PSN, depending upon the PSN implementation.

One model of TCP operation is to imagine that there is a basic GATEWAY associated with each TCP which provides an interface to the local network. This basic GATEWAY performs routing and packet reformatting or embedding, and may also implement congestion and error control between the TCP and GATEWAYS at or intermediate to the destination TCP.

At a GATEWAY between networks, the internetwork packet is unwrapped from its local packet format and examined to determine through which network the internetwork packet should travel next. The internetwork packet is then wrapped in a local packet format suitable to the next network and passed on to a new packet switch.

A GATEWAY is permitted to break up the fragment carried by an internetwork packet into smaller fragments if this is necessary for transmission through the next network. To do this, the GATEWAY produces a set of internetwork packets, each carrying a new fragment. The packet format is designed so that the destination TCP may treat fragments created by the source TCP or by intermediate GATEWAYS nearly identically.

The TCP is responsible for regulating the flow of internetwork packets to and from the processes it serves, as a way of preventing its host from becoming saturated or overloaded with traffic. The TCP is also responsible for retransmitting unacknowledged packets, and for detecting duplicates. A consequence of this error detection/retransmission scheme is that the order of letters received on a given connection is also maintained [CEKA74, SUNS74]. To perform these functions, the TCP opens and closes connections between ports as described in Section 4.3. The TCP performs retransmission, duplicate detection, sequencing, and flow control on all communication among the processes it serves.

2. The TCP INTERFACE to the USER

2.1 The TCP as a POST OFFICE

The TCP acts in many ways like a postal service since it provides a way for processes to exchange letters with each other. It sometimes happens that a process may offer some service, but not know in advance what its correspondents' addresses are. The analogy can be drawn with a mail order house which opens a post office box which can accept mail from any source. Unlike the post box, however, once a letter from a particular correspondent arrives, a port becomes specific to the correspondent until the owner of the port declares otherwise.

In addition to acting like a postal service, the TCP insures end-to-end acknowledgement, error correction, duplicate detection, sequencing, and flow control.

2.2 Sockets and Addressing

We have borrowed the term SOCKET from the ARPANET terminology [CACR70,MCKE73]. In general, a socket is the concatenation of a NETWORK identifier, TCP identifier, and PORT identifier. A CONNECTION is fully specified by the pair of SOCKETS at each end since the same local socket may participate in many connections to different foreign sockets.

Once the connections is specified in the OPEN command [see section 2.3.2], the TCP supplies a [short] Local Connection Name by which the user refers to the connection in subsequent commands. In particular this facilitates using connections with initially unspecified foreign sockets.

TCP's are free to associate ports with processes however they choose. However, several basic concepts seem necessary in any implementation. There must be well known sockets [WKS] which the TCP associates only with the "appropriate" processes by some

means. We envision that processes may "own" sockets, and that processes can only initiate connections on the sockets they own [means for implementing ownership is a local issue, but we envision a Request Port user call, or a method of uniquely allocating a group of ports to a given process, e.g. by associating the high order bits of a port name with a given process.]

Once initiated, a connection may be passed to another process that does not own the local socket [e.g. from logger to service process]. Strictly speaking this is a reconnection issue which might be more elegantly handled by a general reconnection protocol as discussed in section 3.3. To simplify passing a connection within a single TCP, such "invisible" switches may be allowed as in TENEX systems.

Of course, each connection is associated with exactly one process, and any attempt to reference that connection by another process will be signalled as an error by the TCP. This prevents stealing data from or inserting data into another process' data stream.

A connection is initiated by the rendezvous of an arriving internetwork packet and a waiting Transmission Control Block [TCB] created by a user OPEN, SEND, INTERRUPT, or RECEIVE call [see section 2.3]. The matching of local and foreign socket identifiers determines when a successful connection has been initiated. The connection becomes established when sequence numbers have been synchronized in both directions as described in section 4.3.2.

It is possible to specify a socket only partially by setting the PORT identifier to zero or setting both the TCP and PORT identifiers to zero. A socket of all zero is called UNSPECIFIED. The purpose behind unspecified sockets is to provide a sort of "general delivery" facility [useful for logger type processes with well known sockets].

There are bounds on the degree of unspecificity of socket identifiers. TCB's must have fully specified local sockets, although the foreign socket may be fully or partly unspecified. Arriving packets must have fully specified sockets.

We employ the following notation:

x.y.z = fully specified socket with x=net, y=TCP, z=port

x.y.u = as above, but unspecified port

x.u.u = as above, but unspecified TCP and port

u.u.u = completely unspecified

with respect to implementation, u = 0 [zero]

We illustrate the principles of matching by giving all cases of incoming packets which match with existing TCB's. Generally, both the local (foreign) socket of the TCB and the foreign (local) socket of the packet must match.

	TCB local	TCB foreign	Packet local	Packet foreign
(a)	a.b.c	e.f.g	e.f.g	a.b.c
(b)	a.b.c	e.f.u	e.f.g	a.b.c
(c)	a.b.c	e.u.u	e.f.g	a.b.c
(d)	a.b.c	u.u.u	e.f.g	a.b.c

There are no other legal combinations of socket identifiers which match. Case (d) is typical of the ARPANET well known socket idea in which the well known socket (a.b.c) LISTENS for a connection from any (u.u.u) socket. Cases (b) and (c) can be used to restrict matching to a particular TCP or net.

2.3 TCP USER CALLS

2.3.1 A Note on Style

The following sections functionally define the USER/TCP interface. The notation used is similar to most procedure or function calls in high level languages, but this usage is not meant to rule out trap type service calls [e.g. SVC's, UUO's, EMT's,...].

The user calls described below specify the basic functions the TCP will perform to support interprocess communication. Individual implementations should define their own exact format, and may provide combinations or subsets of the basic functions in single calls. In particular, some implementations may wish to automatically OPEN a connection on the first SEND, RECEIVE, or INTERRUPT issued by the user for a given connection.

In providing interprocess communication facilities, the TCP must not only accept commands, but also return information to the processes it serves. This communication consists of:

(a) general information about a connection [interrupts, remote close, binding of unspecified foreign socket].

(b) replies to specific user commands indicating succcess or various types of failure.

Although the means for signalling user processes and the exact format of replies will vary from one implementation to another, it would promote common understanding and testing if a common set of codes were adopted. Such a set of Event Codes is described in section 2.4.

With respect to error messages, references to "local" and "foreign" are ambiguous unless it is known whether these refer to the world as seen by the sender or receiver of the error message. The authors attempted several different approaches and finally settled on the convention that these references would be as seen by the receiver of the message.

2.3.2 OPEN CONNECTION

Format: OPEN(local port, foreign socket [, timeout]) → lcn

We assume that the local TCP is aware of the identity of the processes it serves and will check the authority of the process to use the connection specified. Depending upon the implementation of the TCP, the source network and TCP identifiers will either be supplied by the TCP or by the processes that serve it [e.g. the program which interfaces the TCP to its packet switch or the packet switch itself]. These considerations are the result of concern about security, to the extent that no TCP be able to masquerade as another one, and so on. Similarly, no process can masquerade as another without the collusion of the TCP.

If no foreign socket is specified [i.e. the foreign socket parameter is 0 or not present], then this constititutes a LISTENING local socket which can accept communication from any foreign socket. Provision is also made for partial specification of foreign sockets as described in section 2.2.

If the specified connection is already OPEN, an error is returned, otherwise a full-duplex transmission control block [TCB] is created and partially filled in with data from the OPEN command parameters. The TCB format is described in more detail in section 4.2.2.

No network traffic is generated by the OPEN command. The first

SEND or INTERRUPT by the local user or the foreign user will cause the TCP to synchronize the connection.

The timeout, if present, permits the caller to set up a timeout for all letters transmitted on the connection. If a letter is not successfully transmitted within the timeout period, the user is notified and may ignore the condition [TCP will continue trying to transmit] or direct the TCP to close the connection. The present global default is 30 seconds, and connections which are set up without specifying another timeout will retransmit every letter for at least 30 seconds before notifying the user. The retransmission rate may vary, and is the responsibility of the TCP and not the user. Most likely, it will be related to the measured time for responses to return from letters sent.

Depending on the TCP implementation, either a local connection name will be returned to the user by the TCP, or the user will specify this local connection name (in which case another parameter is needed in the call). The local connection name can then be used as a short hand term for the connection defined by the <local socket, foreign socket> pair.

Responses from the TCP which may occur as a result of this call are detailed in section 2.4.

2.3.3 SEND LETTER

this sends the data in buffer

Format: SEND(local connection name, buffer address, byte count, EOL flag [, timeout])

This call causes the data contained in the indicated user buffer to be sent on the indicated connection. If the connection has not been opened, the SEND is considered an error. Some implementation may allow users to SEND first, in which case an automatic OPEN would be done. If the calling process is not authorized to use this connection, an error is returned.

If the EOL flag is set, the data is the End Of a Letter, and the EOL bit will be set in the last packet created from the buffer. If the EOL flag is not set, subsequent SEND's will appear as part of the same letter. This extended letter facility should be used sparingly because some TCP's may delay processing packets until an entire letter is received.

If no foreign socket was specified in the OPEN, but the connection is established [e.g. because a LISTENing connection has become specific due to a foreign letter arriving for the

local port] then the designated letter is sent to the implied foreign socket. In general, users who make use of OPEN with an unspecified foreign socket can make use of SEND without ever explicitly knowing the foreign socket address.

However, if a SEND is attempted before the foreign socket becomes specified, an error will be returned. Users can use the STATUS call to determine the status of the connection. In some implementations the TCP may notify the user when an unspecified socket is bound.

If the timeout is specified, then the current default timeout for this connection is changed to the new one. This can affect not only all letters sent including and after this one, but also those which have not yet been sent, since the timeout is kept in the TCB and not associated with each letter sent. Of course, a time is maintained for each internetwork packet formed so as to determine how long each of these has been on the retransmission queue.

In the simplest implementation, SEND would not return control to the sending process until either the transmission was complete or the timeout had been exceeded. This simple method is highly subject to deadlocks and is not recommended. [For example both sides of the connection try to do SEND's before doing any RECEIVE's.] A more sophisticated implementation would return immediately to allow the process to run concurrently with network I/O, and, furthermore, to allow multiple SENDs to be in progress concurrently. Multiple SENDs are served in first come, first served order, so the TCP will queue those it cannot service immediately.

NOTA BENE: In order for the process to distinguish among error or success indications for different letters, the buffer address should be returned along with the coded response to the SEND request. We will offer an example event code format in section 2.4, showing the information which should be returned to the calling process.

The semantics of the INTERRUPT call are described later, but this call can have an effect on letters which have been given to the TCP but not yet sent. In particular, all such letters are flushed by the source TCP. Thus one of the responses to a SEND may be "flushed due to interrupt."

Responses from the TCP which may occur as a result of this call are detailed in section 2.4.

2.3.4 RECEIVE LETTER

Format: RECEIVE(local connection name, buffer address, byte count)

This command allocates a receiving buffer associated with the specified connection. If no OPEN precedes this command or the calling process is not authorized to use this connection, an error is returned.

In the simplest implementation, control would not return to the calling program until either a letter was received, or some error occurred, but this scheme is highly subject to deadlocks [see section 2.3.3]. A more sophisticated implementation would permit several RECEIVE's to be outstanding at once. These would be filled as letters arrive. This strategy permits increased throughput, at the cost of a more elaborate scheme [possibly asynchronous] to notify the calling program that a letter has been received.

If insufficient buffer space is given to reassemble a complete letter, an indication that the buffer holds a partial letter will be given; the buffer will be filled with as much data as it can hold.

The remaining parts of a partly delivered letter will be placed in buffers as they are made available via successive RECEIVES. If a number of RECEIVES are outstanding, they may be filled with parts of a single long letter or with at most one letter each. The event codes associated with each RECEIVE will indicate what is contained in the buffer.

To distinguish among several outstanding RECEIVES, and to take care of the case that a letter is smaller than the buffer supplied, the event code is accompanied by both a buffer pointer and a byte count indicating the actual length of the letter received.

The semantics of the INTERRUPT system call are discussed later, but this call can have an effect on outstanding RECEIVES. When the TCP receives an INTERRUPT, it will flush all data currently queued up awaiting receipt by the receiving process. If no data is waiting, but several buffers have been made available by anticipatory RECEIVE commands, these buffers are returned to the process with an error indicating that any data that might have been placed in those buffers has been flushed. This enables the receiving process to synchronize its RECEIVES with the interrupt. That is, the process can distinguish between RECEIVES issued before the receipt of the INTERRUPT and those issued afterwards.

Responses from the TCP which may occur as a result of this call are detailed in section 2.4.

2.3.5 CLOSE CONNECTION

Format: CLOSE(local connection name)

This command causes the connection specified to be closed. If the connection is not open or the calling process is not authorized to use this connection, an error is returned. Any unfilled receive buffers or pending send buffers will be returned to the user with event codes indicating they were aborted due to the CLOSE. Users should wait for event codes for each SEND before closing the connection if they wish to be certain that all letters were successfully delivered.

The user may CLOSE the connection at any time on his own initiative, or in response to various prompts from the TCP [remote close executed, transmission timeout exceeded, destination inaccessible].

Because closing a connection requires communication with the foreign TCP, connections may remain in the closing state for a short time. Attempts to reopen the connection before the TCP replies to the CLOSE command will result in errors.

Responses from the TCP which may occur as a result of this call are detailed in section 2.4.

2.3.6 INTERRUPT

Format: INTERRUPT(local connection name)

A special control signal is sent to the destination indicating an interrupt condition. This facility can be used to simulate "break" signals from terminals or error or completion codes from I/O devices, for example. The semantics of this signal to the receiving process are unspecified. The receiving TCP will signal the interrupt to the receiving process immediately upon receipt, and will also flush any outstanding letters waiting to be delivered. Since it is possible to tell where in the letter stream this command was invoked, it is possible for the receiving TCP to flush only preceding data. The sending TCP will flush any letters pending transmission, returning a special error code to indicate the flush.

If the connection is not open or the calling process is not authorized to use this connection, an error is returned.

Responses from the TCP which may occur as a result of this call are detailed in section 2.4.

2.3.7 STATUS

Format: STATUS(local connection name)

This command returns a data block containing the following information:

local socket, foreign socket, local connection name, receive window, send window, connection state, number of letters awaiting acknowledgement, number of letters pending receipt [including partial ones], default transmission timeout

Depending on the state of the connection, some of this information may not be available or meaningful. If the calling process is not authorized to use this connection, an error is returned. This prevents unauthorized processes from gaining information about a connection.

Responses from the TCP which may occur as a result of this call are detailed in section 2.4.

2.4 TCP TO USER MESSAGES

2.4.1 TYPE CODES

All messages include a type code which identifies the type of user call to which the message applies. Types are:

0 - General message, does not apply to a particular user call

1 - Applies to OPEN

2 - Applies to CLOSE

3 - Applies to INTERRUPT

10 - Applies to SEND

20 - Applies to RECEIVE

30 - Applies to STATUS

2.4.2 MESSAGE FORMAT [notional]

All messages include the following three fields:

Type code

Local connection name

Event code

For message types 0-3 [General, Open, Close, Interrupt] only these three fields are necessary.

For message type 10 [Send] one additional field is necessary:

Buffer address

For message type 20 [Receive] three additional fields are necessary:

Buffer address

Byte count

End-of-letter flag

For message type 30 [Status] additional data might include:

Local socket, foreign socket

Send window [measures buffer space at foreign TCP]

Receive window [measures buffer space at local TCP]

Connection state [see section 4.3.6]

Number of letters awaiting acknowledgement

Number of letters awaiting receipt

Retransmission timeout

2.4.3 EVENT CODES

The event code specifies the particular event that the TCP wishes to communicate to the user.

In addition to the event code, three flags may be useful to classify the event into major categories and facilitate event processing by the user:

E flag: set if event is an error

L/F flag: indicates whether event was generated by Local TCP, or Foreign TCP or network

P/T flag: indicates whether the event is Permanent or Temporary [retry may succeed]

Events are encoded into 8 bits with the high order bits set to indicate the state of the E, L/F, and P/T flags, respectively.

Events specified so far are listed below with their codes and flag settings. A * means a flag does not apply or can take both values for this event. Additional events may be defined in the course of experimentation.

0	0**	general success
1	ELP	connection illegal for this process
2	0F*	unspecified foreign socket has become bound
3	ELP	connection not open
4	ELT	no room for TCB
5	ELT	foreign socket unspecified
6	ELP	connection already open
	EFP	unacceptable SYN [or SYN/ACK] arrived at foreign TCP. Note: This is not a misprint, the local meaning is different from foreign.
7	EFP	connection does not exist at foreign TCP
8	EFT	foreign TCP inaccessible [may have subcases]
9	ELT	retransmission timeout
10	E*P	buffer flushed due to interrupt
11	0F*	interrupt to user
12	**P	connection closing
13	E**	general error
14	E*P	connection reset

Possible events for each message type are as follows:

Type 0[general]: 2,11,12,14

Type 1[open]: 0,1,4,6,13

Type 2[close]: 0,1,3,13

Type 3[interrupt]: 0,1,3,5,7,8,9,12,13

Type 10[send]: 0,1,3,5,7,8,9,10,11,12,13

Type 20[receive]: 0,1,3,10,12,13

Type 30[status]: 0,1,13

Note that events 6(foreign), 7, 8 are generated at the foreign TCP or in the network[s], and these same codes are used in the error field of the internet packet [see section 4.2.1].

3. HIGHER LEVEL PROTOCOLS

3.1 INTRODUCTION

It is envisioned that the TCP will be able to support higher level protocols efficiently. It should be easy to interface existing ARPANET protocols like TELNET and FTP to the TCP.

3.2 WELL KNOWN SOCKETS

At some point, a set of well known 24 bit port numbers must be picked. The type of service associated with the well known ports might include:

(a) Logger

(b) FTP (File transfer protocol)

(c) RJE (Remote job entry)

(d) Host status

(e) TTY Test

(f) HELP - descriptive, interactive system documentation

We reserve well known socket 0 (24 bits of 0) for global messages destined for a particular TCP but not related to any particular connection. We imagine that this socket would be used for unusual TCP synchronization (e.g. RESET ALL) or for testing purposes (e.g. sending letters to TRASHCAN or ECHO). This does not conflict with

the usage that if a socket is 0, it is unspecified, since no user can SEND, CLOSE, or INTERRUPT on socket 0.

3.3 RECONNECTION PROTOCOL (RCP)

Port identifiers fall into two categories: permanent and transient. For example, a Logger process is generally assigned a port identifier that is fixed and well known. Transient processes will in general have ID's which are dynamically assigned.

In the distributed processing environment of the network, two processes that don't have well known port identifiers may often wish to communicate. This can be achieved with the help of a well known process using a reconnection protocol. Such a protocol is briefly outlined using the communication facilities provided by the TCP. It essentially provides a mechanism by which port identifiers are exchanged in order to establish a connection between a pair of sockets.

Such a protocol can be used to achieve the dynamic establishment of new connections in order to have multiple processes solving a problem co-operatively, or to provide a user process access to a server process via a logger, when the logger's end of the connection can not be invisibly passed to the server process.

A paper on this subject by R. Schantz [SCHA74] discusses some of the issues associated with reconnection, and some of the ideas contained therein went into the design of the protocol outlined below.

In the ARPANET, a protocol was implemented which would allow a process to connect to a well known socket, thus making an implicit request for service, and then be switched to another socket so that the well known socket could be freed for use by others. Since sockets in our TCP are permitted to have connections with more than one foreign socket, this facility may not be explicitly needed (i.e. connections <A,B> and <A,C> are distinguishable).

However, the well known socket may be in one network and the actual service socket(s) may be in another network (or at least in another TCP). Thus, the invisible switching of a connection from one port to another within a TCP may not be sufficient as an "Initial Connection Protocol". We imagine that a process wishes to use socket N1.T1.Q to access well known socket N2.T2.P. However, the process associated with socket N2.T2.P will actually start up a new process somewhere which will use N3.T3.S as its server socket. The N(i) and T(i) may be distinct or the same. The user will send to N2.T2.P the relevant user information such as user name, password, and account. The server will start up the server

process and send to N1.T1.Q the actual service socket identifier: N3.T3.S. The connection (N1.T1.Q,N2.T2.P) can then be closed, and the user can do a RECEIVE on (N1.T1.Q,N3.T3.S). The serving process can SEND on (N3.T3.S,N1.T1.Q). There are many variations on this scheme, some involving the user process doing a RECEIVE on a different socket (e.g. (N1.T1.X,U.U.U)) with the server doing SEND on (N3.T3.S,N1.T1.X).

Without showing all the detail of synchronization of sequence numbers and the like, we can illustrate the exchange as shown below.

```
USER                                    SERVER

                                  1.RECEIVE(N2.T2.P,U.U.U)

1. SEND(N1.T1.Q,N2.T2.P)==>

                          <== 2. SEND(N2.T2.P,N1.T1.Q)

                                    with "N3.T3.S" as data

2. RECEIVE(N1.T1.Q,N2.T2.P)

3. CLOSE(N1.T1.Q,N2.T2.P)==>

                          <== 3. CLOSE(N2.T2.P,N1.T1.Q)

4. RECEIVE(N1.T1.Q,N3.T3.S)

                          <== 4. SEND(N3.T3.S,N1.T1.Q)
```

At this point, a connection is open between N1.T1.Q and N3.T3.S. A variation might be to have the user do an extra RECEIVE on (N1.T1.X,U.U.U) and have the data "N1.T1.X" be sent in the first user SEND. Then, the server can start up the real serving process and do a SEND on (N3.T3.S,N1.T1.X) without having to send the "N3.T3.S" data to the user. Or perhaps both server and receiver exchange this data, to assure security of the ultimate connection (i.e. some wild process might try to connect to N1.T1.X if it is merely RECEIVING on foreign socket U.U.U.).

We do not propose any specific reconnection protocol here, but leave this to further deliberation, since it is really a user level protocol issue.

4. TCP IMPLEMENTATION

4.1 INTRODUCTION

Conceptually, the TCP is made up of several processes. Some of these deal with USER/TCP commands, and others with packets arriving from the network. The TCP also has an internal measurement facility which can be activated remotely.

Any particular TCP could be viewed in a number of ways. It could be implemented as an independent process, servicing many user processes. It could be viewed as a set of re-entrant library routines which share a common interface to the local PSN, and common buffer storage. It could even be viewed as a set of processes, some handling the user, some the input of packets from the net, and some the output of packets to the net.

4.2 TCP DATA STRUCTURES

4.2.1 INTERNETWORK PACKET FORMAT

8 bits: Internet information

2 bits: Reserved for local PSN use

2 bits: Header format (11 in binary)

4 bits: Protocol version number

8 bits: Header length in octets (32 is the current value)

16 bits: Length of text in octets

32 bits: Packet sequence number

32 bits: Acknowledgement number (i.e. sequence number of next octet expected).

16 bits: Window size (in octets)

16 bits: Control Information

Listed from high to low order:

SYN: Request to synchronize sending sequence numbers

ACK: There is a valid Acknowledgement in the 32 bit ACK field

FIN: Sender will stop SENDing and RECEIVEing on this connection

DSN: The sender has stopped using sequence numbers and wants to initiate a new sequence number for sending.

EOS: This packet is the end of a segment and therefore has a checksum in the 16 bit checksum field. If this bit is not set, the 16 bit checksum field is to be ignored. The bit is usually set, but if fragmentation at a GATEWAY occurs, the packets preceding the last one will not have checksums, and the last packet will have the checksum for the entire original fragment (segment) as it was calculated by the sending TCP.

EOL: This packet contains the last fragment of a letter. The EOS bit will always be set in this case.

INT: The sender wants to INTERRUPT on this connection.

XXX: six (6) unused control bits

CD: three (3) bits of control dispatch:

000: Null (the control octet contents should be ignored)

001: Event Code is present in the control octet. These were defined in section 2.4.3.

010: Special Functions

011: Reject (codes as yet undefined)

1XX: Unused

8 bits: Control Data Octet

If CD is 000 then this octet is to be ignored.

If CD is 001, this octet contains event codes defined in section 2.4.3

If CD is 010, this octet contains a special function code as defined below:

0: RESET all connections between Source and Detination TCPs

1: RESET the specific connection referenced in this packet

2: ECHO return packet to sender with the special function code ECHOR (Echo Reply).

3: QUERY Query status of connection referenced in this packet

4: STATUS Reply to QUERY with requested status.

5: ECHOR Echo Reply

6: TRASH Discard packet without acknowledgement

>6: Unused

Note: Special function packets not pertaining to a particular connection [RESET all, ECHO, ECHOR, and TRASH] are normally sent using socket zero as described in section 3.2.

If CD is 011, this octet contains an as yet undefined REJECT code.

If CD is 1XX, this octet is undefined.

4 bits: Length of destination network address in 4 bit units (current value is 1)

4 bits: Destination network address

1010-1111 are addresses of ARPANET, UCL, CYCLADES, NPL, CADC, and EPSS respectively.

16 bits: Destination TCP address

8 bits: Padding

4 bits: Length of source network address in 4 bit units (current value is 1)

4 bits: Source network address (as for destination address).

16 bits: Source TCP address

24 bits: Destination port address

24 bits; Source port address

16 bits: Checksum (if EOS bit is set)

4.2.2 TRANSMISSION CONTROL BLOCK

It is highly likely that any implementation will include shared data structures among parts of the TCP and some asynchronous means of signalling users when letters have been delivered.

One typical data structure is the Transmission Control Block (TCB) which is created and maintained during the lifetime of a given connection. The TCB contains the following information (field sizes are notional only and may vary from one implementation to another):

16 bits: Local connection name

48 bits: Local socket

48 bits: Foreign socket

16 bits: Receive window size in octets

32 bits: Receive left window edge (next sequence number expected)

16 bits: Receive packet buffer size of TCB (may be less than window)

16 bits: Send window size in octets

32 bits: Send left window edge (earliest unacknowledged octet)

32 bits: Next packet sequence number

16 bits: Send packet buffer size of TCB (may be less than window)

8 bits: Connection state

E/C - 1 if TCP has been synchronized at least once (i.e. has been established, else 0, meaning it is closed; this bit is reset after FINs are exchanged and the user has done a CLOSE). The bit is not reset if the connection is only desynchronized on send or receive or both directions.

SS - SYNCed on send side (if set) else desynchronized

SR - SYNCed on receive side (if set, else desynchronized)

16 bits: Special flags

S1 - SYN sent if set

S2 - SYN verified if set

R - SYN received if set

F - FIN sent if set

C - CLOSE from local user received if set

U - Foreign socket unspecified if set

SDS - Send side DSN sent if set

SDV - Send side DSN verified if set

RDR - Receive side DSN received if set

Initially, all bits are off [no pun intended] (i.e. SS, SR, E/C, S1, S2, R, F, C, SDS, SDV, RDR =0) When R is set, so is SR. When S1 and S2 are both set, so is SS. SR is reset when RDR is set. SS is reset when both SDS and SDV are set. These bits are used to keep track of connection state and to aid in arriving packet processing (e.g. Can sequence number be validated? Only if SR is set.).

16 bits: Retransmission timeout (in eigths of a second)

16 bits: Head of Send buffer queue [buffers SENT from user to TCP, but not packetized]

16 bits: Tail of Send buffer queue

16 bits: Pointer to last octet packetized in partially packetized buffer (refers to the buffer at the head of the queue)

16 bits: Head of Send packet queue

16 bits: Tail of Send packet queue

16 bits: Head of Packetized buffer queue

16 bits: Tail of Packetized buffer queue

16 bits: Head of Retransmit packet queue

16 bits: Tail of Retransmit packet queue

16 bits: Head of Receive buffer queue [queue of buffers given by user to RECEIVE letters, but unfilled]

16 bits: Tail of Receive buffer queue

16 bits: Head of Receive packet queue

16 bits: Tail of receive packet queue

16 bits: Pointer to last contiguous receive packet

16 bits: Pointer to last octet filled in partly filled buffer

16 bits: Pointer to next octet to read from partly emptied packet

[Note: The above two pointers refer to the head of the receive buffer and receive packet queues respectively]

16 bits: Forward TCB pointer

16 bits: Backward TCB pointer

4.3 CONNECTION MANAGEMENT

4.3.1 INITIAL SEQUENCE NUMBER SELECTION

The protocol places no restriction on a particular connection being used over and over again. New instances of a connection will be referred to as incarnations of the connection. The problem that arises owing to this is, "how does the TCP identify duplicate packets from previous incarnations of the connection?". This problem becomes harmfully apparent if the connection is being opened and closed in quick succession, or if the connection breaks with loss of memory and is then reestablished.

The essence of the solution [TOML74] is that the initial sequence number [ISN] must be chosen so that a particular sequence number can never refer to an "old" octet. Once the connection is established the sequencing mechanism provided by the TCP filters out duplicates.

For an association to be established or initialized, the two TCP's must synchronize on each others initial sequence numbers. Hence the solution requires a suitable mechanism for picking an

initial sequence number [ISN], and a slightly involved handshake to exchange the ISN's. A "three way handshake" is necessary because sequence numbers are not tied to a global clock in the network, and TCP's may have different mechanisms for picking the ISN's. The receiver of the first SYN has no way of knowing whether the packet was an old delayed one or not, unless it remembers the last sequence number used on the connection which is not always possible, and so it must ask the sender to verify this SYN.

The "three way handshake" and the advantages of a "clock-driven" scheme are discussed in [TOML74]. More on the subject, and algorithms for implementing the clock-driven scheme can be found in [DALA74].

4.3.2 ESTABLISHING A CONNECTION

The "three way handshake" is essentially a unidirectional attempt to establish the connection, i.e. there is an initiator and a responder. The TCP's should however be able to establish the connection even if a simultaneous attempt is made by both TCP's to establish the connection. Simultaneous attempts are treated like "collisions" in "Aloha" systems and these conflicts are resolved into unidirectional attempts to establish the connection. This scheme was adopted because

(i) Connections will normally have a passive and an active end, and so the mechanism should in most cases be as simple as possible.

(ii) It is easy to implement as special cases do not have to be accounted for.

The example below indicates what a three way handshake between TCP's A and B looks like

A		B
-->	<SEQ x><SYN>	-->
<--	<SEQ y><SYN, ACK x+1>	<--
-->	<SEQ x+1><ACK y+1><DATA BYTES>	-->

The receiver of a "SYN" is able to determine whether the "SYN" was real (and not an old duplicate) when a positive "ACK" is returned for the receiver's "SYN,ACK" in response to the "SYN". The sender of a "SYN" gets verification on receipt of a

Postel

"SYN,ACK" whose "ACK" part references the sequence number proposed in the original "SYN" [pun intended].

If the TCP is in the state where it is waiting for a response to its SYN, but gets a SYN instead, then it always thinks this is a collision and goes into the state prior to having sent the SYN, i.e. it forgets that it had sent a SYN. The TCP will try to establish the connection again after some time, unless it has to respond to an arriving SYN. Even if the wait times in the two TCPs are the same, the varying delays in network transmission will usually be adequate to avoid a collision on the next cycle of attempts to send SYN.

When establishing a connection, the state of the TCP is represented by 3 bits --

S1 S2 R

S1 = 1 -- SYN sent

S2 = 1 -- My SYN verified

R = 1 -- SYN received

Some examples of attempts to establish the connection are now shown. The state of the connection is indicated when a change occurs. We specifically do not show the cases in which connection synchronization is carried out with packets containing both SYN and data. We do this to simplify the explanation, but we do not rule out an implementation which is capable of dealing with data arriving in the first packet (it has to be stored temporarily without acknowledgement or delivery to the user until the arriving SYN has been verified).

The "three way handshake" now looks like --

```
S1S2R  A                                   B  S1S2R

0 0 0                                         0 0 0

     --> <SEQ x><SYN>              -->

1 0 0                                         0 0 1

     <-- <SEQ y><SYN, ACK x+1>     <--

1 1 1                                         1 0 1

     --> <SEQ x+1><ACK y+1><DATA OCTETS>-->
```

```
1 1 1                                                    1 1 1
```

The scenario for a simultaneous attempt to establish the connection without the arrival of any delayed duplicates is --

```
     S1S2R  A                                      B  S1S2R

     0 0 0                                           0 0 0

(M1)  1 0 0--> <SEQ x><SYN>                          ...

(M2)  0 0 0<-- <SEQ y><SYN>                          <-- 1 0 0

(M1)           B returns to no SYN sent              --> 0 0 0

(M1)  1 0 0--> <SEQ z><SYN>   *                      --> 0 0 1

(M3)  1 1 1<-- <SEQ y+1><SYN,ACK z+1>                <-- 1 0 1

(M4)  1 1 1--> <SEQ z+1><ACK y+1><DATA>              --> 1 1 1
```

Note: "..." means that a message does not arrive, but is delayed in the network. State changes are upon arrival or upon departure of a given message, as the case may be. Packets containing the SYN or INT or DSN bits implicitly contain a "dummy" data octet which is never delivered to the user, but which causes the packet sequence numbers to be incremented by 1 even if no real data is sent. This permits the acknowledgement of these controls without acknowledging receipt of any data which might also have been carried in the packet. A packet containing a FIN bit has a dummy octet following the last octet of data (if any) in the packet.

* Once in state 000 sender selects new ISN z when attempting to establish the connection again.

4.3.3 HALF-OPEN CONNECTIONS

An established connection is said to be a "half-open" connection if one of the TCP's has closed the connection at its end without the knowledge of the other, or if the two ends of the connection have become desynchronized owing to a crash that resulted in loss of memory. Such connections will automatically become reset if an attempt is made to send data in either direction. However, half-open connections are expected to be unusual, and the recovery procedure is somewhat involved.

If one end of the connection no longer exists, then an attempt by the other user to send any data on it will result in the

sender receiving the event code "Connection does not exist at foreign TCP". Such an error message should indicate to the user process that something is wrong and it is expected to CLOSE the connection.

Assume that two user processes A and B are communicating with one another when a crash occurs causing loss of memory to B's TCP. Depending on the operating system supporting B's TCP, it is likely that some error recovery mechanism exists. When the TCP is up again B is likely to start again from the beginning or from a recovery point. As a result B will probably try to OPEN the connection again or try to SEND on the connection it believes open. In the latter case it receives the error message "connection not open" from the local TCP. In an attempt to establish the connection B's TCP will send a packet containing SYN. A's TCP thinks that the connection is already established and so will respond with the error "unacceptable SYN (or SYN/ACK) arrived at foreign TCP". B's TCP knows that this refers to the SYN it just sent out, and so should reset the connection and inform the user process of this fact.

It may happen that B is passive and only wants to receive data. In this case A's data will not reach B because the TCP at B thinks the connection is not established. As a result A's TCP will timeout and send a QRY to B's TCP. B's TCP will send STATUS saying the connection is not synced. A's TCP will treat this as if an implicit CLOSE had occurred and tell the user process, A, that the connection is closing. A is expected to respond with a CLOSE command to his TCP. However, A's TCP does not send a FIN to B's TCP, since it would not be accepted anyway on the unsynced connection. Eventually A will try to reopen the connection or B will give up and CLOSE. If B CLOSEs, B's TCP will simply delete the connection since it was not established as far as B's TCP is concerned. No messages will be sent to A's TCP as a result.

4.3.4 RESYNCHRONIZING A CONNECTION

Details of resynchronization have not yet been specified since the need for this should be infrequent in the initial testing stages.

4.3.5 CLOSING A CONNECTION

There are essentially three cases:

a) The user initiates by telling the TCP to CLOSE the connection.

b) The remote TCP initiates by sending a FIN control signal

c) Both users CLOSE simultaneously

Two bits are used to maintain control over the closing of a connection: these are called the "FIN sent" bit [F] and the "USER Closed" bit, [C] respectively. The control procedure uses these two bits to assure that the connection is properly closed.

Case 1: Local user initiates the close

In this case, both the F and C bits are initially zero, but the C bit is set immediately upon receipt of the user call "CLOSE." When the FIN is sent out by the TCP, the F bit is set. All pending RECEIVES are terminated and the user is told that they have been prematurely terminated ("connection closing") without data. Similarly, any pending SENDS are terminated with the same response, "connection closing."

Several responses may arrive as the result of sending a FIN. The one which is generally expected is a matching FIN. When this is received, the TCB can be eliminated. If a "connection does not exist at foreign TCP" message comes in response to the FIN, then the TCB can likewise be eliminated. If no response is forthcoming, or if "Foreign TCP inaccessible" arrives then the resolution is moot. One might simply timeout and discard the TCB. Since the local user wants to CLOSE anyway, this is probably satisfactory, although it will leave a potential "half-open" connection at the other side. We deal with half open connections in section 4.3.3.

When the acknowledging FIN arrives after the connection state bits are set (F=1, C=1), then the TCB can be deleted.

Case 2: TCP receives a FIN from the network

First of all, a FIN must have a sequence number which lies in the valid receive window. If not, it is discarded and the left window edge is sent as acknowledgement. If the FIN can be processed, it is handled (possibly out of order, since it is taken as an imperative to shut down the connection). All pending RECEIVES and SENDS are responded to by showing that they were terminated by the other side's close request (i.e. "connection closing"). The user is also told by an unsolicited event or signal that the connection has been closed (in some systems, the user might have to request

STATUS to get this information). Finally, the TCP sends FIN in response.

Thus, because a FIN arrived, a FIN is sent back, so the F bit is set. However, the TCB stays around until the local user does a CLOSE in acknowledgement of the unsolicited signal that the connection has been closed by the other side. Thus, the C bit remains unset until this happens. If the C and F bits go from (F=1 C=0) to (F=1, C=1), then the connection is closed and the TCB can be removed.

case 3: both users close simultaneously

If this happens, both connections will be in the (F=1, C=1) state. When the FINs arrive, the connections will be shut down. If one FIN fails to arrive, we have two choices. One is to insist on acknowledgements for FINS, in which case the missing one will be retransmitted. Another is merely to permit the half-open connection to remain (we prefer this solution). It can timeout independently and go away after a while. If an attempt is made to reestablish the connection, the initiator will discover the existence of the old connection since an "inappropriate SYN received" message will be sent by the TCP which holds the "half-open" connection. The receiver of this message can tell the other TCP to reset the connection. We cannot permit the holder of the half-open connection to reset automatically on receipt of the SYN since its receipt is not necessarily prima facie evidence of a half open connection. (The SYN could be a delayed duplicate.)

4.3.6. CONNECTION STATE and its relation to USER and INCOMING CONTROL REQUESTS

In order to formalize the action taken by the TCP when it receives commands from the User, or Control information from the network, we define a connection to be in one of 7 states at any instant. These are known as the TCB Major States. Each Major State is simply a convenient name for a particular setting or group of settings of the state bits, as follows:

S1	S2	R	U	F	C	#	name
-	-	-	-	-	-	0	no TCB
0	0	0	0/1	0	0	1	unsync
1	0	0	0	0	0	2	SYN sent
1	0	1	0/1	0	0	3	SYN received
1	1	1	0	0	0	4	established
1	0/1	1	0/1	1	1	5	FIN wait
1	1	1	0	1	0	6	FIN received

The connection moves from state to state as shown below. The transition from one state to another will be represented as

[X, Y]<cause><action>

which means that there is a transition from state X to state Y owing to <cause>. The action taken by the TCP is specified as <action>. We use this notation to give the important state transitions, often simplifying the cause and action fields to take into account a number of situations. Figure 1 illustrates these transitions in traditional state diagram form. Section 4.4.6 and section 4.4.7 fully specify the effect of all User commands and Control information arriving from the network.

[0,1] <OPEN> <create TCB>

[1,2] <SEND,INTERRUPT, or collision timeout> <send SYN>

[1,3] <SYN arrives> <send SYN,ACK>

[1,0] <CLOSE> <remove TCB>

[2,1] <SYN arrives (collision)> <set timeout, forget SYNs>

[2,0] <CLOSE> <remove TCB>

[2,4] <appropriate SYN,ACK arrives> <send ACK>

[3,4] <appropriate ACK arrives> <none>

[3,1] <error arrives or timeout> <(forget SYN)>

[3,5] <CLOSE> <send FIN>

[4,5] <CLOSE> <send FIN>

[4,6] <appropriate FIN arrives> <send FIN, inform user>

[5,0] <FIN or error arrives, or timeout> <remove TCB>

[6,0] <CLOSE> <remove TCB>

4.4 STRUCTURE OF THE TCP

4.4.1 INTRODUCTION [See figure 2.1]

There are many possible implementations of the TCP. We offer one conceptual framework in which to view the various algorithms that make up the TCP design. In our concept, the TCP is written in two parts, an interrupt or signal driven part (consisting of four processes), and a reentrant library of subroutines or system calls which interface the user process to the TCP. The subroutines communicate with the interrupt part through shared data structures (TCB's, shared buffer queues etc.). The four processes are the Output Packet Handler which sends packets to the packet switch; the Packetizer which formats letters into internet packets; the Input Packet Handler which processes incoming packets; and the Reassembler which builds letters for users.

The ultimate bottleneck is the pipe through which arriving and departing packets must travel. This is the Host/Packet Switch interface. The interrupt driven TCP shares among all TCB's its limited packet buffer resources for sending and receiving packets. From the standpoint of controlling buffer congestion, it appears better to treat incoming packets with higher priority than outgoing packets. That is, packet buffers which can be released by copying their contents into user buffers clearly help to reduce congestion. Neither the packetizer nor the input packet handler should be allowed to take up all available packet buffer space; an analogous problem arises in the IMP in the allocation of store and forward, and reassembly buffer space. One policy is to permit neither contender more than, say, two-thirds of the space. The buffer allocation routines can enforce these limits and reject buffer requests as needed. Conceptually, the scheduler can monitor the amounts of storage dedicated to the input and output routines, and can force either to sleep if its buffer allocation exceeds the limit.

As an example, we can consider what happens when a user

executes a SEND call to the TCP service routines. The buffer containing the letter is placed on a SEND buffer queue associated with the user's TCB. A 'packetizer' process is awakened to look through all the TCB's for 'packetizing' work. The packetizer will keep a roving pointer through the TCB list which enables it to pick up new buffers from the TCB queue and packetize them into output buffers. The packetizer takes no more than one letter at a time from any single TCB. The packetizer attempts to maintain a non-empty queue of output packets so that the output handler will not fall idle waiting for the packetizing operation. However, since arriving packets compete with departing packets, care must be taken to prevent either class from occupying all of the shared packet buffer space. Similarly, since the TCB's all compete for space in service to their connections, neither input nor output packet space should be dominated by any one TCB.

When a packet is created, it is placed on a FIFO SEND packet queue associated with its origin TCB. The packetizer wakes the output handler and then continues to packetize a few more buffers, perhaps, before going to sleep. The output handler is awakened either by a 'hungry' packet switch or by the packetizer; in either case, it uses a roving TCB pointer to select the next TCB for service. The send packet queue can be used as a 'work queue' for the output handler. After a packet has been sent, but usually before an ACK is returned, the output handler moves the packet to a retransmission queue associated with each TCB.

Retransmission timeouts can refer to specific packets and the retransmission list can be searched for the specific packet. If an ACK is received, the retransmission entry can be removed from the retransmit queue. The send packet queue contains only packets waiting to be sent for the first time. INTERRUPT requests can remove entries in both the send packet queue and the retransmit packet queue.

Since packets are never in more than one queue at a time, it appears possible for INT, FIN or RESET commands to remove packets from the receive, send, or retransmit packet queues with the assurance that an already issued signal to either the reassembler, the packetizer or the output handler will not be confusing.

Handling the INTERRUPT and CLOSE functions can however require some care to avoid confusing the scheduler, and the various processes. The scheduler must maintain status information for the processes. This information includes the current TCB being serviced. When an INTERRUPT is issued by a local process, the

output queue of letters associated with the local port referenced is to be deleted. The packetizer, for example, may however be working at that time on the same queue. As usual, simultaneous reading and writing of the TCB queue pointers must be inhibited through some sort of semaphore or lockout mechanism. When the packetizer wants to serve the next send buffer queue, it must lock out all other access to the queue, remove the head of the queue (assuming of course that there are enough buffers for packetization), advance the head of the queue, and then unlock access to the queue.

If the packetizer keeps only a TCB pointer in a global place called CPTCB (current packetizer TCB address), and always uses the address in CPTCB to find the TCB in which to examine the send buffer queue, then removal of the output buffer queue does not require changes to any working storage belonging to the packetizer. Even more important, the arrival and processing of a RESET or CLOSE, which clears the system of a given TCB, can update the CPTCB pointer, as long as the removal does not occur while the packetizer is still working on the TCB.

Incoming packets are examined by the input packet handler. Here they are checked for valid connection sockets and acknowledgements are processed, causing packets to be removed, possibly, from the SEND or RETRANSMIT packet queues, as needed. As an example, consider the receipt of a valid FIN request on a particular TCB. If a FIN had not been sent before (i.e. F bit not set), then a FIN packet is constructed and sent after having cleared out the SEND buffer and SEND packet queues as well as the RETRANSMIT queue. Otherwise, if the F and C bits are both set, all queues are emptied and the TCB is returned to free storage.

Packets which should be reassembled into letters and sent to users are queued by the input packet handler, on the receive packet queue, for processing by the reassembly process. The reassembler looks at its FIFO work queue and tries to move packets into user buffers which are queued up in an input buffer queue on each TCB. If a packet has arrived out of order, it can be queued for processing in the correct sequence. Each time a packet is moved into a user buffer, the left window edge of the receiving TCB is moved to the right so that outgoing packets can carry the correct ACK information. If the SEND buffer queue is empty, then the reassembler creates a packet to carry the ACK.

As packets are moved into buffers and they are filled, the buffers are dequeued from the RECEIVE buffer queue and passed to the user. The reassembler can also be awakened by the

RECEIVE user call should it have a non-empty receive packet queue with an empty RECEIVE buffer queue. The awakened reassembler goes to work on each TCB, keeping a roving pointer, and sleeping if a cycle is made of all TCB's without finding any work.

4.4.2 INPUT PACKET HANDLER [See figure 2.2]

The Input Packet Handler is awakened when a packet arrives from the network. It first verifies that the packet is for an existing TCB (i.e. the local and foreign socket numbers are matched with those of existing TCB's). If this fails, an error message is constructed and queued on the send packet queue of a dummy TCB. A signal is also sent to the output packet handler. Generally, things to be transmitted from the dummy TCB have a default retransmission timeout of zero, and will not be retransmitted. (We use the idea of a dummy TCB so that all packets containing errors, or RESET can be sent by the output packet handler, instead of having the originator of them interface to the net. These packets, it will be noticed, do not belong to any TCB).

The input packet handler looks out for control or error information and acts appropriately. Section 4.4.7 discusses this in greater detail, but as an example, if the incoming packet is a RESET request of any kind (i.e. all connections from designated TCP or given connection), and is believable, then the input packet handler clears out the related TCB(s), empties the send and receive packet queues, and prepares error returns for outstanding user SEND(s) and RECEIVE(s) on each reset TCB. The TCB's are marked unused and returned to storage. If the RESET refers to an unknown connection, it is ignored.

Any ACK's contained in incoming packets are used to update the send left window edge, and to remove the ACK'ed packets from the TCB retransmit packet queue. If the packet being removed was the end of a user buffer, then the buffer must be dequeued from the packetized buffer queue, and the User informed. The packetizer is also signalled. Only one signal, or one for each packet, will have to be sent, depending on the scheduling scheme for the processes. See section 4.4.7 for a detailed discussion.

The packet sequence number, the current receive window size, and the receive left window edge determine whether the packet lies within the window or outside of it.

Let W = window size

S = size of sequence number space

L = left window edge

R = L+W-1 = right window edge

x = sequence number to be tested

For any sequence number, x, if

(R-x) mod S <= W

then x is within the window.

A packet should be rejected only if all of it lies outside the window. This is easily tested by letting x be, first the packet sequence number, and then the sum of packet sequence number and packet text length, less one. If the packet lies outside the window, and there are no packets waiting to be sent, then the input packet handler should construct a dummy ACK and queue it for output on the send packet queue, and signal the output packet handler. Successfully received packets are placed on the receive packet queue in the appropriate sequence order, and the reassembler signalled.

The packet window check can not be made if the associated TCB is not in the 'established' state, so care must be taken to check for control and TCB state before doing the window check.

4.4.3 REASSEMBLER [See figure 2.3]

The Reassembler process is activated by both the Input Packet Handler and the RECEIVE user call. While the reassembler is asleep, if multiple signals arrive, all but one can be discarded. This is important as the reassembler does not know the source of the signal. This is so in order that "dangling" signals from work in TCB's that have subsequently been removed don't confuse it. Each signal simply means that there may be work to be done. If the reassembler is awake when a signal arrives, it may be necessary to put it in a "hyperawake" state so that even if the reassembler tries to quit, the scheduler will run it one more time.

When the reassembler is awakened it looks at the receive packet queue for each TCB. If there are some packets there then it sees whether the RECEIVE buffer queue is empty. If it is then the reassembler gives up on this TCB and goes on to the next one, otherwise if the first packet matches the left window edge, then the packet can be moved into the User's buffer. The

reassembler keeps transferring packets into the User's buffer until the letter is completely transferred, or something causes it to stop. Note that a buffer may be partly filled and then a sequence 'hole' is encountered in the receive packet queue. The reassembler must mark progress so that the buffer can be filled up starting at the right place when the 'hole' is filled. Similarly a packet might be only partially emptied when a buffer is filled, so progress in the packet must be marked.

If a letter was successfully transferred to a User buffer then the reassembler signals the User that a letter has arrived and dequeues the buffer associated with it from the TCB RECEIVE buffer queue. If the buffer is filled then the User is signaled and the buffer dequeued as before. The event code indicates whether the buffer contains all or part of a letter, as described in section 2.4.

In every case when a packet is delivered to a buffer, the receive left window edge is updated, and the packetizer is signaled. This updating must take account of the extra octet included in the sequencing for certain control functions [SYN, INT, FIN, DSN]. If the send packet queue is empty then the reassembler must create a packet to carry the ACK, and place it on the send packet queue.

Note that the reassembler never works on a TCB for more than one User buffer's worth of time, in order to give all TCBs equal service.

Scheduling of the reassembler is a big issue, but perhaps running to completion will be satisfactory, or else it can be time sliced. In the latter case it will continue from where it left off, but a new signal may have arrived producing some possible work. This work will be processed as part of the old incomplete signal, and so some wasteful processing may occur when the reassembler wakes up again. This is the general problem of trying to implement a protocol that is fundamentally asynchronous, but at least it is immune to harmful race conditions. e.g. If we were to have the reassembler 'remove' the signal that caused it to wake up, just before it went to sleep (in order that new arriving ones were discarded) then a new signal may arrive at a critical time causing it not to be recognised: thus leaving some work pending, and this may result in a deadlock. [see previous comments on "hyperawake" state]

4.4.4 PACKETIZER [See figure 2.4]

The Packetizer process gets work from both the Input Packet Handler and the SEND user call. The signal from the SEND user

call indicates that there is something new to send, while the one from the input packet handler indicates that more TCP buffers may be available from delivered packets. This latter signal is to prevent deadlocks in certain kind of scheduling schemes. We assume the same treatment of signals as discussed in section 4.4.3.

When the packetizer is awakened it looks at the SEND buffer queue for each TCB. If there is a new or partial letter awaiting packetization, it tries to packetize the letter, TCP buffer and window permitting. It packetizes no more than one letter for a TCB before servicing another TCB. For every packet produced it signals the output packet handler (to prevent deadlock in a time sliced scheduling scheme). If a 'run till completion ' scheme is used then one signal only need be produced, the first time a packet is produced since awakening. If packetization is not possible the packetizer goes on to the next TCB.

If a partial buffer was transferred then the packetizer must mark progress in the SEND buffer queue. Completely packetized buffers are dequeued from the SEND buffer queue, and placed on a Packetized buffer queue, so that the buffer can be returned to the user when an ACK for the last bit is received.

When the packetizer packetizes a letter it must see whether it is the first piece of data being sent on the connection, in which case it must include the SYN bit. Some implementations may not permit data to be sent with SYN and others may discard any data received with SYN.

The packetizer goes to sleep if it finds no more work at any TCB.

4.4.5 OUTPUT PACKET HANDLER [See figure 2.5]

When activated by the packetizer, or the input packet handler, or some of the user call routines, the Output Packet Handler attempts to transmit packets on the net (may involve going through some other network interface program). It looks at the TCB's in turn, transmitting some packets from the send packet queue. These are dequeued and put on the retransmit queue along with the time when they should be retransmitted.

All data packets that are transmitted have the latest receive left window edge in the ACK field. Error and control messages may have no ACK [ACK bit off], or set the ACK field to refer to a received packet's sequence number.

The RETRANSMIT PROCESS:

This process can either be viewed as a separate process, or as part of the output packet handler. Its implementation can vary; it could either perform its function, by being woken up at regular intervals, or when the retransmission time occurs for every packet put on the retransmit queue. In the first case the retransmit queue for each TCB is examined to see if there is anything to retransmit. If there is, a packet is placed on the send packet queue of the corresponding TCB . The output packet handler is also signaled.

Another "demon" process monitors all user Send buffers and retransmittable control messages sent on each connection, but not yet acknowledged. If the global retransmission timeout is exceeded for any of these, the user is notified and he may choose to continue or close the connection. A QUERY packet may also be sent to ascertain the state of the connection [this facilitates recovery from half open connections as described in section 4.3.3].

4.4.6 USER CALL PROCESSING

OPEN [See figure 3.1]

1. If the process calling does not own the specified local socket, return with <type 1><ELP 1 "connection illegal for this process">.

2. If no foreign socket is specified, construct a new TCB and add it to the list of existing TCB's. Select a new local connection name and return it along with <type 1><OLP 0 "success">. If there is no room for the TCB, respond with <type 1><ELT 4 "No room for TCB">.

3. If a foreign socket is specified, verify that there is no existing TCB with the same <local socket, foreign socket> pair (i.e. same connection), otherwise return <type 1><ELP 6 "connection already open">. If there is no TCB space, return as in (2), otherwise, create the TCB and link it with the others, returning a local connection name with the success event code.

Note: if a TCB is created, be sure to copy the timeout parameter ino it, and set the "U" bit to 0 if a foreign socket is specified, else set U to 1 (to show unspecified foreign socket).

SEND [See figure 3.2]

1. Search for TCB with local connection name specified. If none found, return <type 10><ELP 3 "connection not open">.

2. If TCB is found, check foreign socket specification. If not set (i.e. U = 1 in TCB), return <type 10><ELT 5 "foreign socket unspecified">. If the connection is in the "closing" state (i.e. state 5 or 6), return <type 3><ELP 12 "connection closing"> and do not process the buffer.

3. Put the buffer on the Send Buffer queue and signal the packetizer that there is work to do.

INTERRUPT [See figure 3.3]

1. Validate existence of the referenced connection, sending out error messages of the form <type 3><ELP 3 "connection not open"> or <type 3><ELT 5 "foreign socket unspecified"> as appropriate. If the local connection refers to a connection not accessible to the process interupting, send <type 3><ELP 1 "connection illegal for this process">.

2. If the connecction is in the "closing" state (i.e. states 5 or 6), return <type 3><ELT 12 "connection closing"> and do not send an INT packet to the destination.

3. Any pending SEND buffers should be returned with <type 10><ELP 10 "buffer flushed due to interrupt">. An INT packet should be created and placed on the ouput packet queue, and the output packet handler should be signalled.

RECEIVE [See figure 3.4]

1. If the caller does not have access to the referenced local connection name, return <type 20><ELP 1 "connection illegal for this process">. And if the connection is not open, return <type 20><ELP 3 "connection not open">. If the connection is in the closing state (e.g. a FIN has been received or a user CLOSE is being processed), return <type 20><ELP 12 "connection closing">.

2. Otherwise, put the buffer on the receive buffer queue and signal the reassembler that buffer space is available.

CLOSE [See figure 3.5]

1. If the connection is not accessible to the caller, return <type 2><ELP 1 "connection illegal for this process">. If there is no such connection respond with <type 2><ELP 3 "connection not open">.

2. If the R bit is 0 (i.e. connection is in state 1 or 2), simply remove the TCB.

3. If the R bit is set and the F bit is set, then remove the TCB.

4. Otherwise, if the R bit is set, but F is 0 (i.e. states 3 or 4), return all buffers to the user with <type x><ELP 12 "connection closing">, clear all output and input packet queues for this connection, create a FIN packet, and signal the output packet handler. Set the C and F bits to show this action.

STATUS [See figure 3.6]

1. If the connection is illegal for the caller to access, send <type 30><ELP 1 "connection illegal for this process">.

2. If the connection does not exist, return <type 30><ELP 3 "connection not open">.

3. Otherwise get status information from the TCB and return it via <type 30><0-T 0 "status data...">.

4.4.7 NETWORK CONTROL PROCESSING

The Input Packet Handler examines the header to see if there is any control information or error codes present. We do not discuss the action taken for various special function codes, as it is often implementation dependent, but we describe those that affect the state of the connection. After initial screening by the IPC [see section 4.4.2 and figure 2.2], control and error packets are processed as shown in figures 4.1-4.7. [ACK and data processing is done within the IPC.]

4.4.8 TCP ERROR HANDLING

Error messages have CD=001 and do not carry user data. Depending on the error, zero or more octets of error information will be carried in the packet text field. We explicitly assume that this data is restricted in length so as to fall below the GATEWAY fragmentation threshold (probably 512 bits of data and header). Errors generally refer to specific connections, so the source and destination socket identifiers are relavent here. The ACK field of an error packet contains the sequence number of the packet that caused the error, and the ACK bit is off. [RESET and STATUS special functions may use the ACK field in the same way.] This allows the receiver of an

error message to determine which packet caused the error. Error packets are not ACK'ed or retransmitted.

4.5. BUFFER AND WINDOW ALLOCATION

4.5.1 INTRODUCTION

The TCP manages buffer and window allocation on connections for two main purposes: equitably sharing limited TCP buffer space among all connections (multiplexing function), and limiting attempts to send packets, so that the receiver is not swamped (flow control function). For further details on the operation and advantages of the window mechanism see CEKA74.

Good allocation schemes are one of the hardest problems of TCP design, and much experimentation must be done to develop efficient and effective algorithms. Hence the following suggestions are merely initial thoughts. Different implementations are encouraged with the hope that results can be compared and better schemes developed.

Several of the measurements discussed in a later section are aimed at providing information on the performance of allocation mechanisms. This should aid in determining significant parameters and evaluating alternate schemes.

4.5.2 The SEND Side

The window is determined by the receiver. Currently the sender has no control over the SEND window size, and never transmits beyond the right window edge. There exists the possibility of specifying two more special function codes so that the sender can request the receiver to INCREASE or DECREASE the window size, without specifying by how much. The receiver, of course, needn't satisfy this request.

Buffers must be allocated for outgoing packets from a TCP buffer pool. The TCP may not be willing to allocate a full window's worth of buffers, so buffer space for a connection may be less than what the window would permit. No deadlocks are possible even if there is insufficient buffer or window space for one letter, since the receiver will ACK parts of letters as they are put into the user's buffer, thus advancing the window and freeing buffers for the remainder of the letter.

It is not mandatory that the TCP buffer outgoing packets until acknowledgements for them are received, since it is possible to reconstruct them from the actual letters sent by the user.

However, for purposes of retransmission and processing efficiency it is very convenient to do.

4.5.3 The RECEIVE Side

At the receiving side there are two requirements for buffering:

(1) Rate Discrepancy:

If the sender produces data much faster or much slower than the receiver consumes it, little buffering is needed to maintain the receiver at near maximum rate of operation. Simple queueing analysis indicates that when the production and consumption (arrival and service) rates are similar in magnitude, more buffering is needed to reduce the effect of stochastic or bursty arrivals and to keep the receiver busy.

(2) Disorderly Arrivals:

When packets arrive out of order, they must be buffered until the missing packets arrive so that packets (or letters) are delivered in sequence. We do not advocate the philosophy that they be discarded, unless they have to be, otherwise a poor effective bandwidth may be observed. Path length, packet size, traffic level, routing, timeouts, window size, and other factors affect the amount by which packets come out of order. This is expected to be a major area of investigation.

The considerations for choosing an appropriate window are as follows:

Suppose that the receiver knows the sender's retransmission timeout, 'K'. This is usually close to the round trip transmission time. Suppose also, that the receiver's acceptance rate is 'U' bits/sec, and the window size is 'W' bits. Ignoring line errors and other traffic, the sender transmits at a rate between W/K and the maximum line rate (the sender can send a window's worth of data each timeout period).

If W/K is greater than U, the difference must be retransmissions which is undesirable, so the window should be reduced to W', such that W'/K is approximately equal to U. This may mean that the entire bandwidth of the transmission channel is not being used, but it is the fastest rate at which the receiver is accepting data, and the line capacity is free for other users. This is exactly the same case where the rates of the sender and receiver were almost equal, and so more

buffering is needed. Thus we see that line utilization and retransmissions can be traded off against buffering.

If the receiver does not accept data fast enough (by not performing sufficient RECEIVES) the sender may continue retransmitting since unaccepted data will not be ACK'ed. In this case the receiver should reduce the window size to "throttle" the sender and inhibit useless retransmissions.

Receiver window control:

If the user at the receiving side is not accepting data, the window should be reduced to zero. In particular, if all TCP incoming packet buffers for a connection are filled with received packets, the window must go to zero to prevent retransmissions until the user accepts some packets.

Short term flow control:

Let F = the number of user receive buffers filled

B = the total user receive buffers

W = the long-term or nominal window size

W' = the window size returned to the sender

then a possible value for W' is

W' = W*[1-F/B]**a

The value of 'a' should be greater than one, in order to shut the window faster as buffers run out. The values of W' and F actually used could be averages of recent values, in order to get smooth control. Note that W' is constantly being recomputed, while the value of W, which sets the upper limit of W', only changes slowly in response to other factors.

The value of W can be large (up to half the sequence number space) to allow for good throughput on high delay channels. The sender needn't allocate W worth of buffer space anyway. The long-term variation of W to match flow requirements may be a separate question

This short-term mechanism for flow control allows some buffering in the two TCP's at either end, (as much as they are willing), and the rest in the user process at the send side where the data is being created. Hence the cost of

buffering to smooth out bursty traffic is borne partly by the TCP's, and partly by the user at the send side. None of it is borne by the communication subnet.

5. NETWORK MEASUREMENT PLANS FOR TCP

5.1 USERLEVEL DIAGNOSTICS

We have in mind a program which will exercise a given TCP, causing it to cycle through a number of states: opening, closing, and transmitting on a variety of connections. This program will collect statistics and will generally try to detect deviation from TCP functional specifications. Clearly there will have to be a copy of this program both at the local site being tested and some site which has a certified TCP. So we will have to produce a specification for this user level diagnostic program also.

There needs to be a master and a slave side to all this so the master can tell the slave what's going wrong with the test.

5.2 SINGLE CONNECTION MEASUREMENTS

Round trip delay times

Time from moment the packet is sent by the TCP to the time that the ACK is received by the TCP.

Time from the moment the USER issues the SEND to the time that the USER gets the successful return code.

Note: Packet size should be used to distingish from one set of round trip times and another.

Network destination, and current configuration and traffic load may also be issues of importance that must be taken into account.

What if the destination TCP decides to queue up ACKs and send a single ACK after a while? How does this affect round trip statistics?

What about out of order arrivals and the bunched ACK for all of them?

The histogram of round trip times include retransmission times and these must be taken into account in the analysis and evaluation of the collected data.

Packet size statistics

Histogram of packet length in both directions on the full duplex connection.

Histogram of letter size in both directions.

Measure of disorderly arrival

Distance from the first octet of arriving packet to the left window edge. A histogram of this measure gives an idea of the out of order nature of packet arrivals. It will be 0 for packets arriving in order.

Retransmission Histogram

Effective throughput

This is the effective rate at which the left edge of the window advances. The time interval over which the measure is made is a parameter of the measurement experiment. The shorter the interval, the more bursty we would expect the measure to be.

It is possible to measure effective data throughput in both directions from one TCP by observing the rate at which the left window edge is moving on ACK sent and received for the two windows.

Since throughput is largely dependent upon buffer allocation and window size, we must record these values also. Varying window for a fixed file transmission might be a good way to discover the sensitivity of throughput to window size.

Output measurment

The throughput measurement is for data only, but includes retransmission. The output rate should include all octets transmitted and will give a measure of retransmission overhead. Output rate also includes packet format overhead octets as well as data.

Utilization

The effective throughput divided by the output rate gives a measure of utilization of the communication connection.

Window and buffer allocation measurements

Histogram of letters outstanding, measured at the instant of SEND receipt by TCP from user or at instant of arrival of a letter for a receiving user.

Buffers in use on the SEND side upon packet departure into the net: buffers in use on the RECEIVE side upon delivery of a packet into a USER buffer.

5.3 MULTICONNECTION MEASUREMENTS

Statistics on User Commands sent to the local TCP

Statistics of error or success codes returned [histogram of each type of error or return response]

Statistics of control bit use

Counter for each control bit over all packets emitted by the TCP and another for packets accepted

Count data carrying packets

Count ACK packets with no data

Error packets distribution by error type code received from the net and sent out into the net

5.4 MEASUREMENT IMPLEMENTATION PHILOSOPHY

We view the measurement process as something which occurs internal to the TCP but which is controllable from outside. A well known socket owned by the TCP can be used to accept control which will select one or more measurement classes to be collected. The data would be periodically sent to a designated foreign socket which would absorb the data for later processing, in the manner currently used in the ARPANET IMPs. Each measurement class has its own data packet format to make the job of parsing and analyzing the data easier.

We would restrict access to TCP measurement control to a few designated sites [e.g. NMC, SU-DSL, BBN]. This is easily done by setting up listening control connections on partially specified foreign sockets.

6. SCHEDULE OF IMPLEMENTATION

7. REFERENCES

1. CEKA74

V. Cerf and R. Kahn, "A Protocol For Packet Network Intercommunication," IEEE Transactions on Communication, vol. C-20, No. 5, May 1974, pp. 637-648.

2. CERF74

V. Cerf, "An Assessment of ARPANET Protocols," in Proceedings of the Jerusalem Conference on Information Technology, July 1974 [RFC#635, INWG Note # ***].

3. CESU74

V. Cerf and C. Sunshine, "Protocols and Gateways for the Interconnection of Packet switching Networks," Proc. of the Subconference on Computer Nets, Seventh Hawaii International Conference on Systems Science, January 1974.

4. HEKA70

F. Heart, R.E. Kahn, et al, "The Interface Message Processor for the ARPA Computer Network," AFIPS 1970 SJCC Proceedings, vol. 36, Atlantic City, AFIPS Press, New Jersey, pp. 551-567.

5. POUZ74

L. Pouzin, "CIGALE, the packet switching machine of the CYCLADES computer network," Proceedings of the IFIP74 Congress, Stockholm, Sweden.

6. ROWE74

L. Roberts and B. Wessler, "Computer Network Development to achieve resource sharing," AFIPS 1970, SJCC Proceedings, vol. 36, Atlantic City, AFIPS Press, New Jersey, pp. 543-549.

7. POUZ73

L. Pouzin, "Presentation and major design aspects of the CYCLADES Computer Network," Data Networks: Analysis and Design, Third Data Communications Symposium, St. Petersburg, Florida, November 1973, PP. 80-87.

8. SCWI71

R. Scantlebury and P.T. Wilkinson, "The Design of a Switching System to allow remote Access to Computer Services by other computers and Terminal Devices," Second Symposium on Problems in the Optimization of Data Communication Systems Proceedings, Palo Alto, California, October 1971, pp. 160-167.

9. POST72

J. Postel, "Official Initial Connection Protocol," Current Network

Protocols. Network Information Center. Stanford Research Institute. Menlo Park. California. January 1972 (NIC 7101).

10. CACR70

C.S. Carr. S.D. Crocker. and V.G. Cerf. "Host-Host Communication Protocol in the ARPA Network." AFIPS Conference Proceedings. vol. 36. 1970 SJCC. AFIPS Press. Montvale. N.J.

11. ZIEL74

H. Zimmerman and M. Elie. "Transport Protocol. Standard Host-Host Protocol for heterogeneous computer networks." INWG#61. April 1974.

12. CRHE72

S. D. Crocker. J. F. Heafner; R. M. Metcalfe and J. B. Postel. "Function-oriented protocols for the ARPA Computer Network." AFIPS Conference Proceedings. vol. 41. 1972 FJCC. AFIPS Press. Montvale. N.J.

13. DALA74

Y. Dalal. "More on selecting sequence numbers." INWG Protocol Note #4. October 1974.

14. SUNS74

C. Sunshine. "Issues in communication protocol design -- formal correctness." INWG Protocol Note #5. October 1974.

15. BELS74

D. Belsnes. "Note on single message communication." INWG Protocol Note #3. September 1974.

16. TOML74

R. Tomlinson. "Selecting sequence numbers." INWG Protocol Note #2. September 1974.

17. SCHA74

R. Schantz. "Reconnection Protocol". private communication; available from Schantz at BBN.

18. POUZ74A

L. Pouzin, "A proposal for interconnecting packet switching networks," INWG Note #60, March 1974 [also submitted to EUROCOMP 74].

19. DLMG74

D. Lloyd, M. Galland, and P. T. Kirstein, "Aims and objectives of internetwork experiments," to be published as an INWG Experiments Note.

20. MCKE73

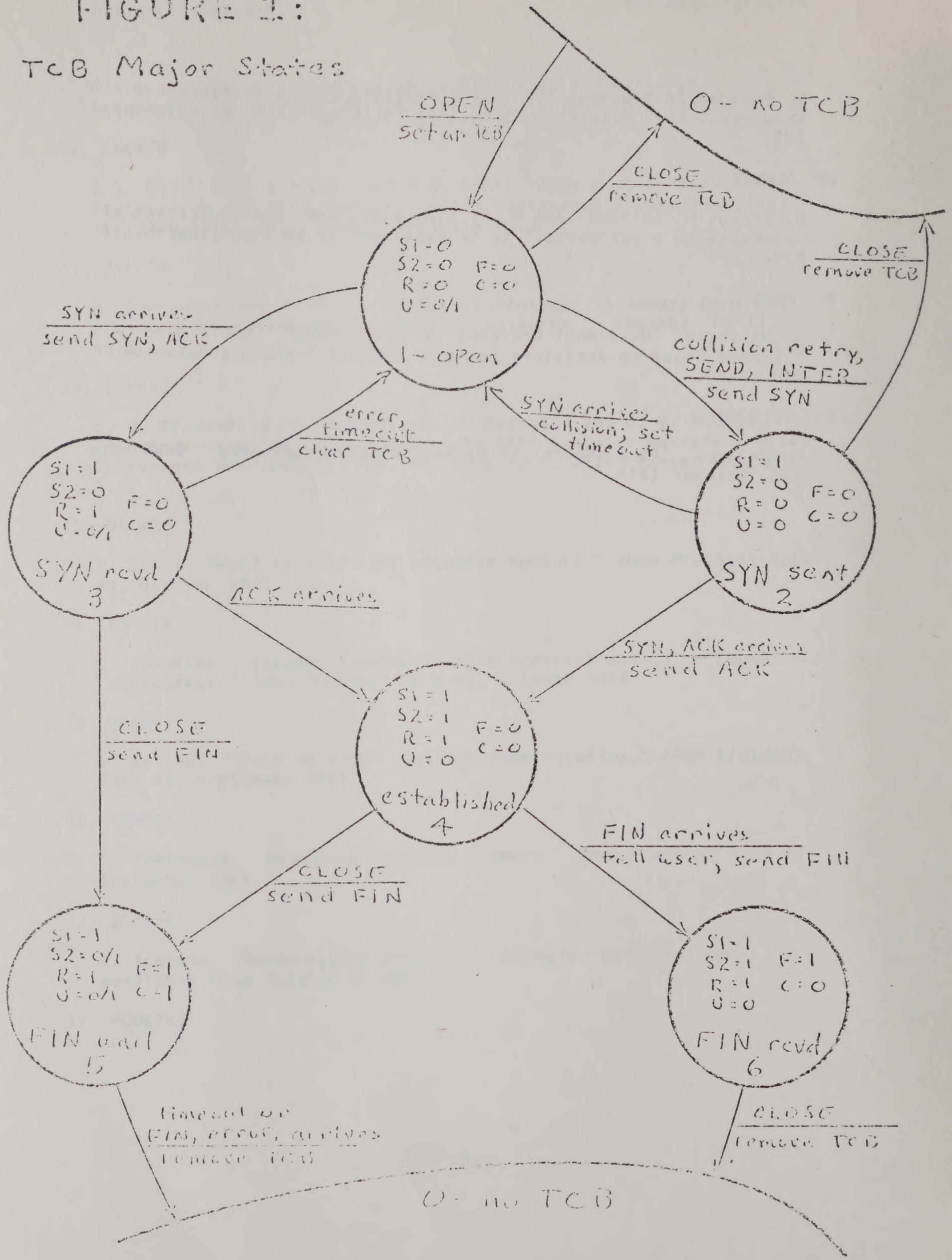
A. Mckenzie, "Host-Host Protocol for the ARPANET," NIC # 8246, Stanford Research Institute [also in ARPANET Protocols Notebook NIC 7104].

21. BELS74A

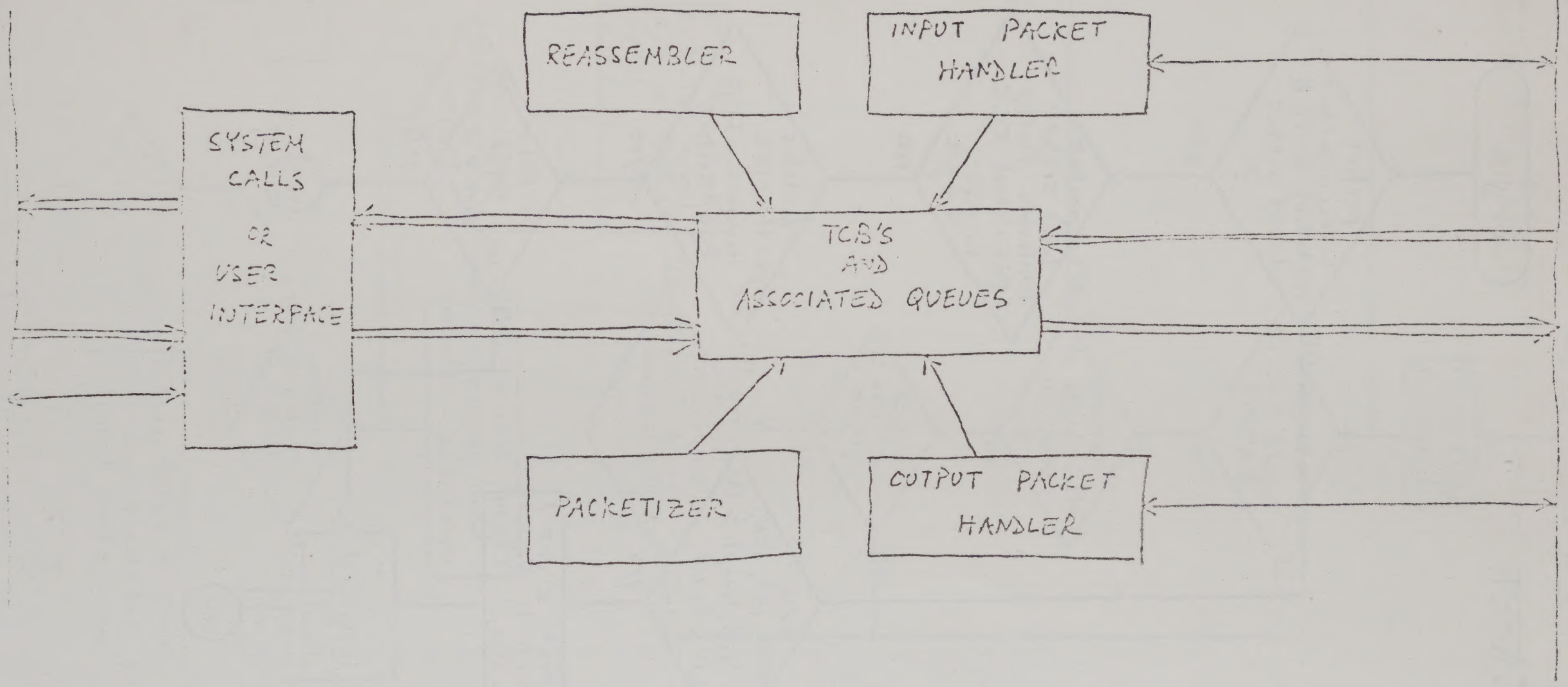
D. Belsnes, "Flow control in packet switching networks," INWG Note #63, October 1974.

FIGURE 1:

TCB Major States



STRUCTURE OF THE TCP



⟹ LOGICAL OR PHYSICAL FLOW OF DATA (PACKETS/LETTERS)

⟶ 'INTERACTION'

NOTE : THE SIGNALLING OF PROCESSES BY OTHERS IS NOT SHOWN

FIG. 2.1

FIGURE 2.2a

Address Check

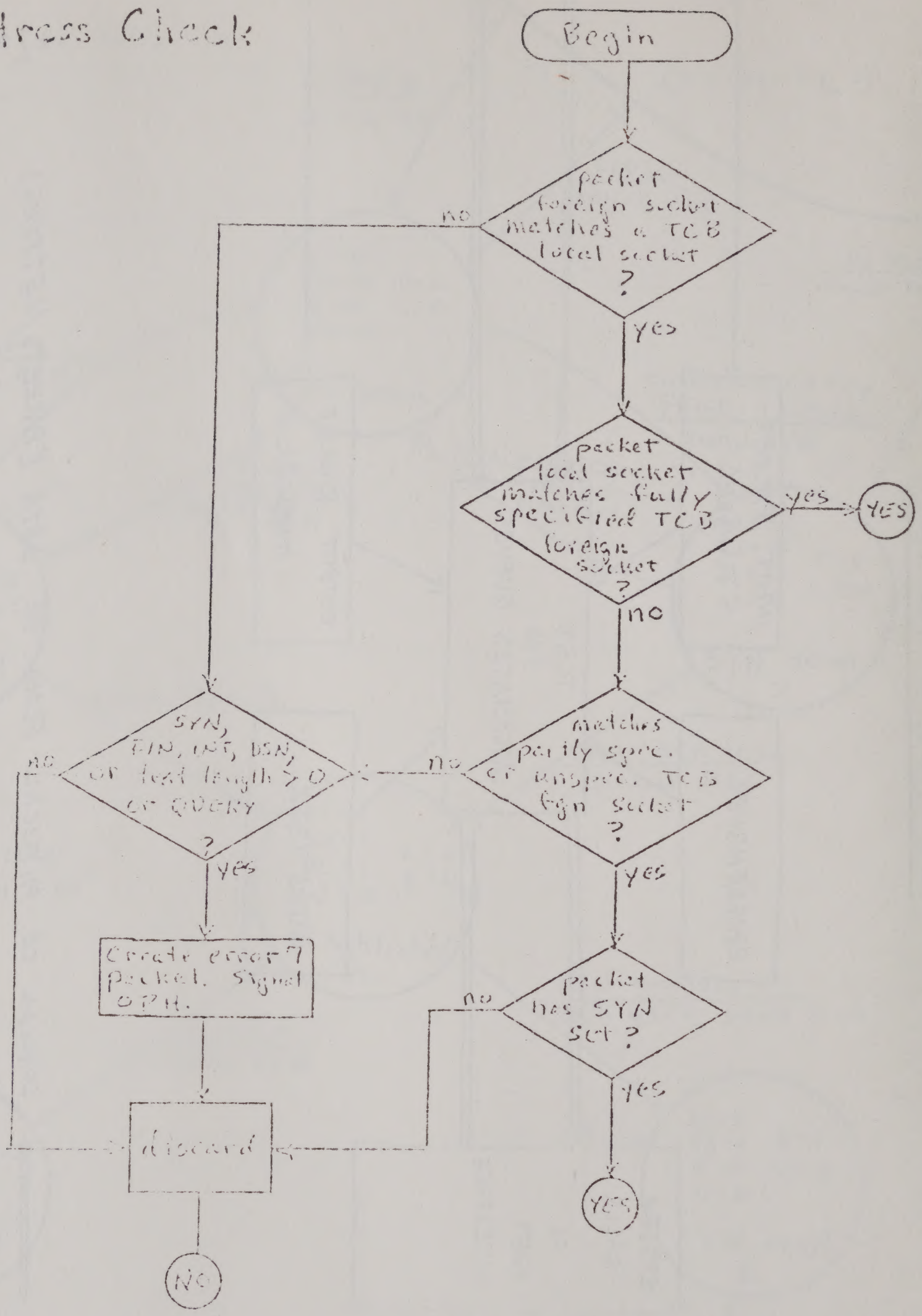


FIGURE 2.3
Reassembler

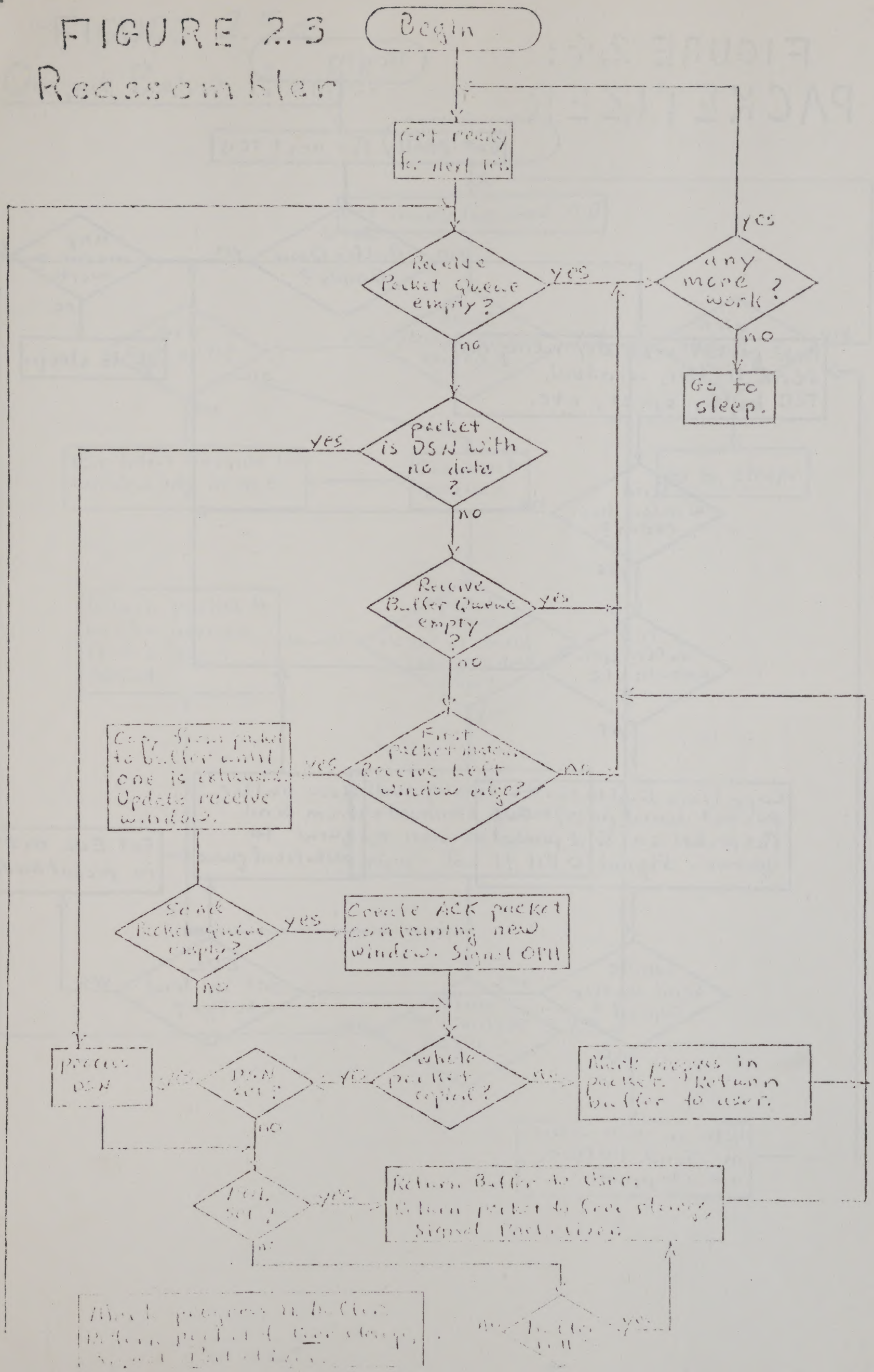


FIGURE 2.4: PACKETIZER

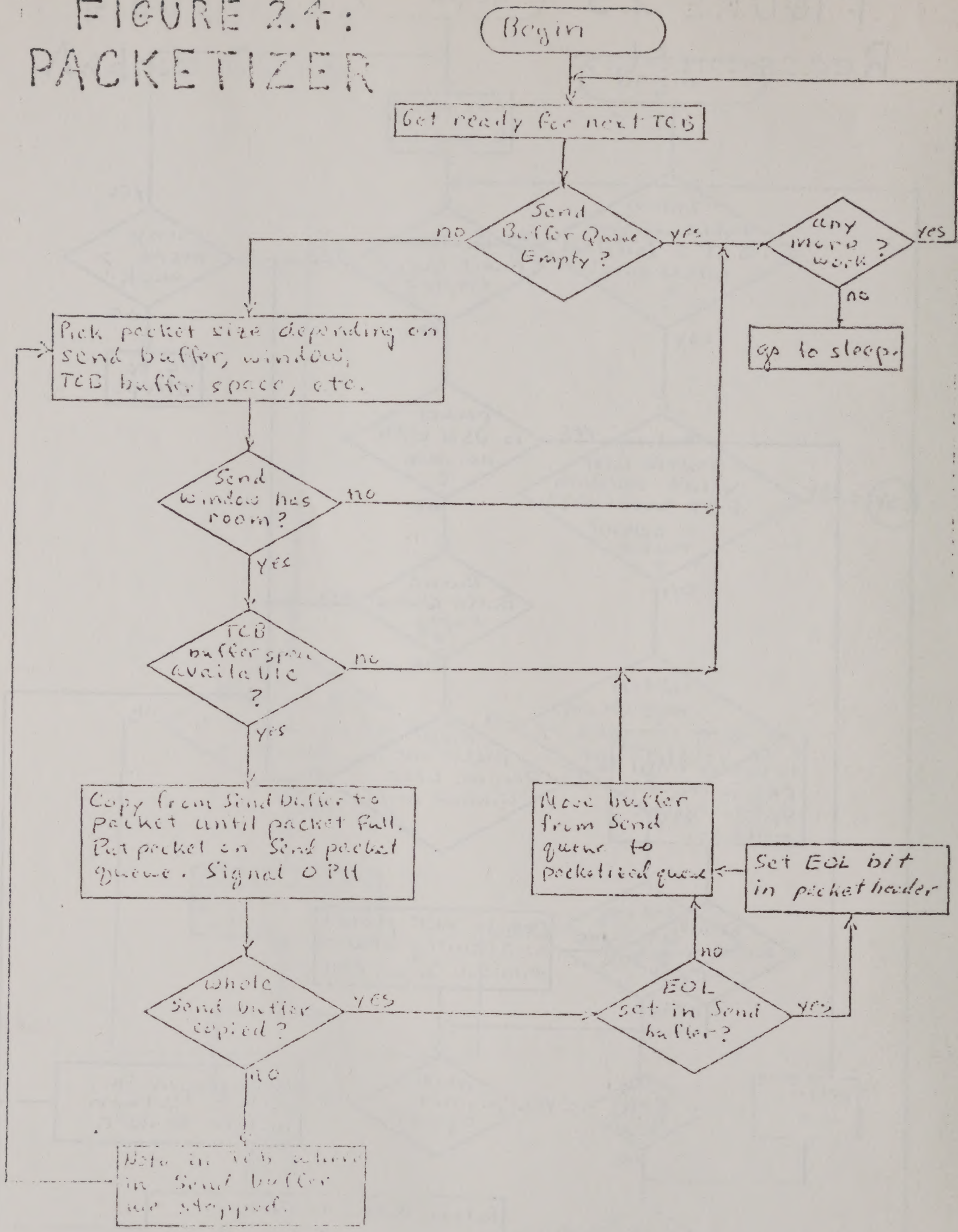

Begin
Get ready for next TCB
Send Buffer Queue Empty ?
no
yes
any more work ?
yes
no
go to sleep.
Pick packet size depending on send buffer, window, TCB buffer space, etc.
Send window has room ?
no
yes
TCB buffer space available ?
no
yes
Copy from Send buffer to packet until packet full. Put packet on Send packet queue. Signal OPH
Move buffer from Send queue to packetized queue
Set EOL bit in packet header
Whole Send buffer copied ?
yes
no
EOL set in Send buffer ?
yes
no
Data in TCB above in Send buffer was stripped.

FIGURE 2.5a

Output Packet Handler

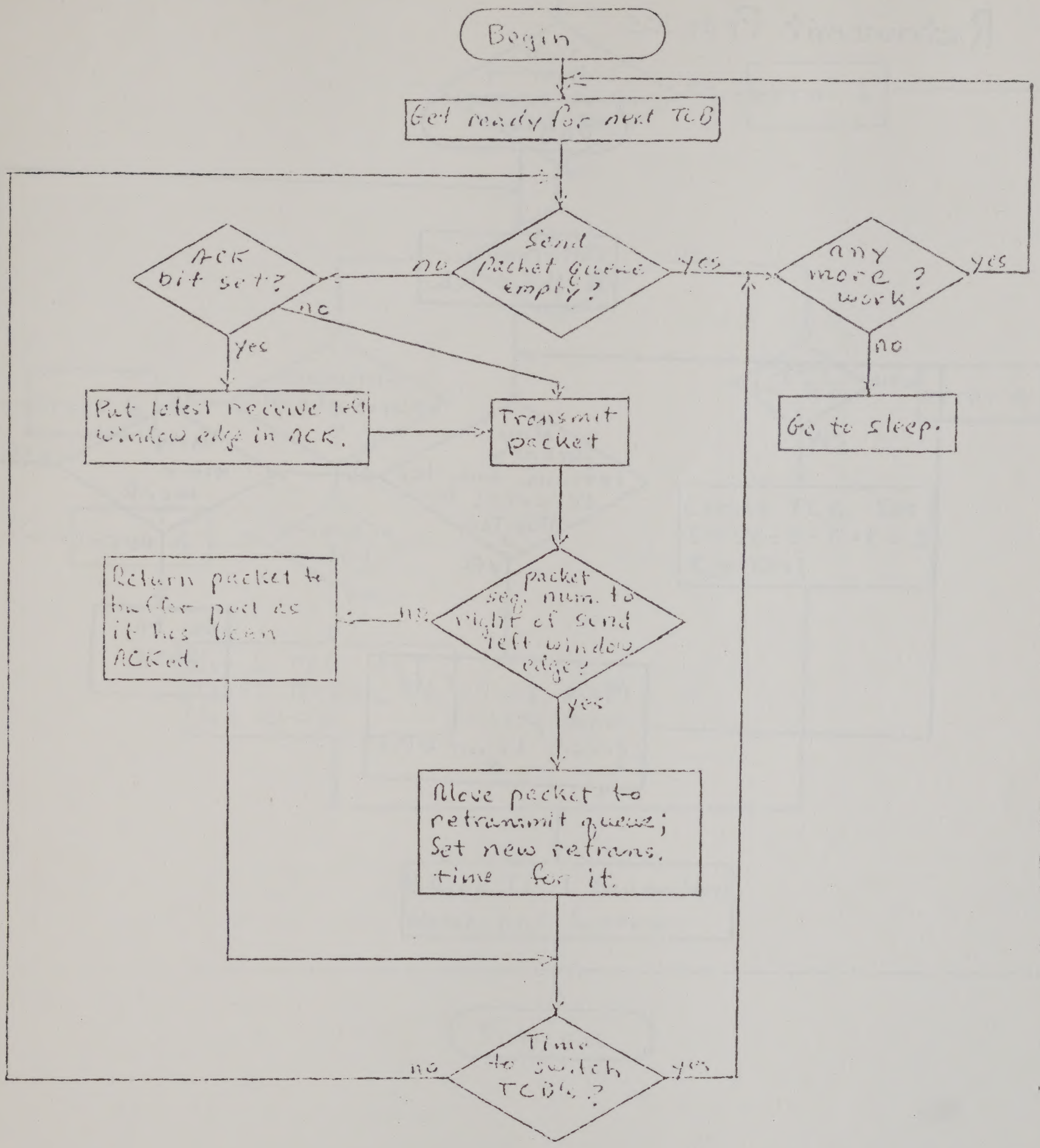


FIGURE 2.5b

Retransmit Process

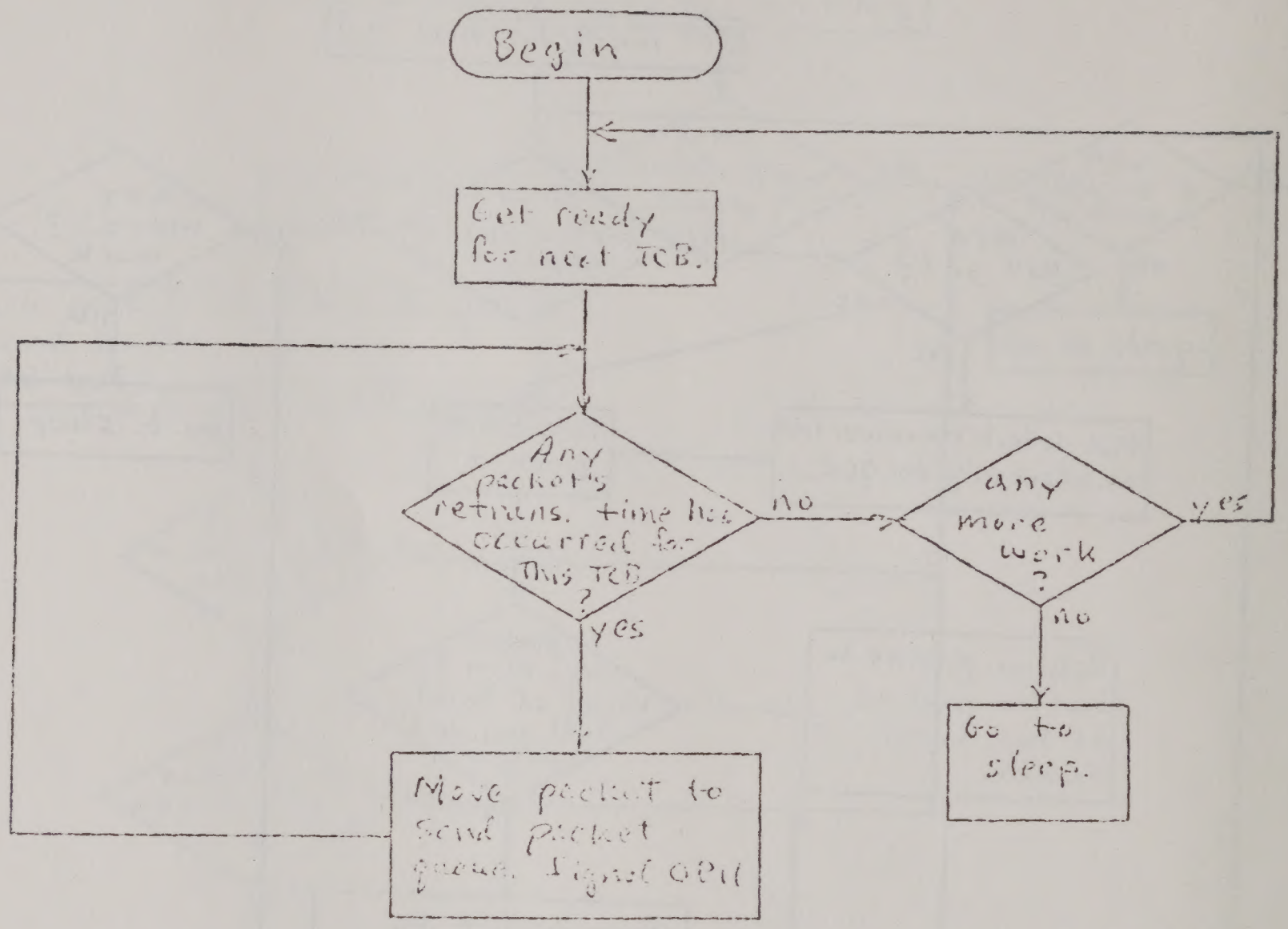


FIGURE 3.1:

OPEN

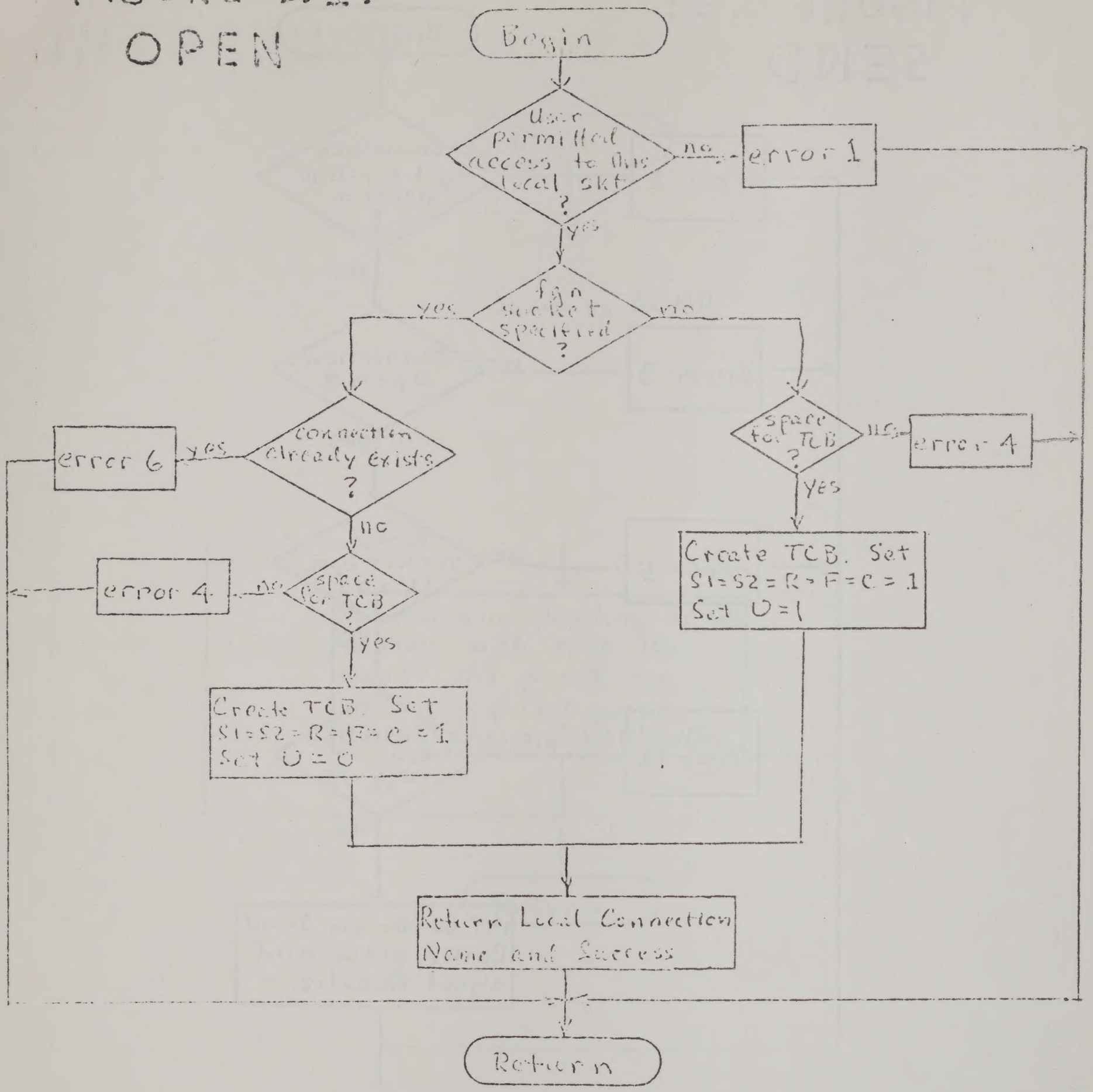


FIGURE 3.2:
SEND

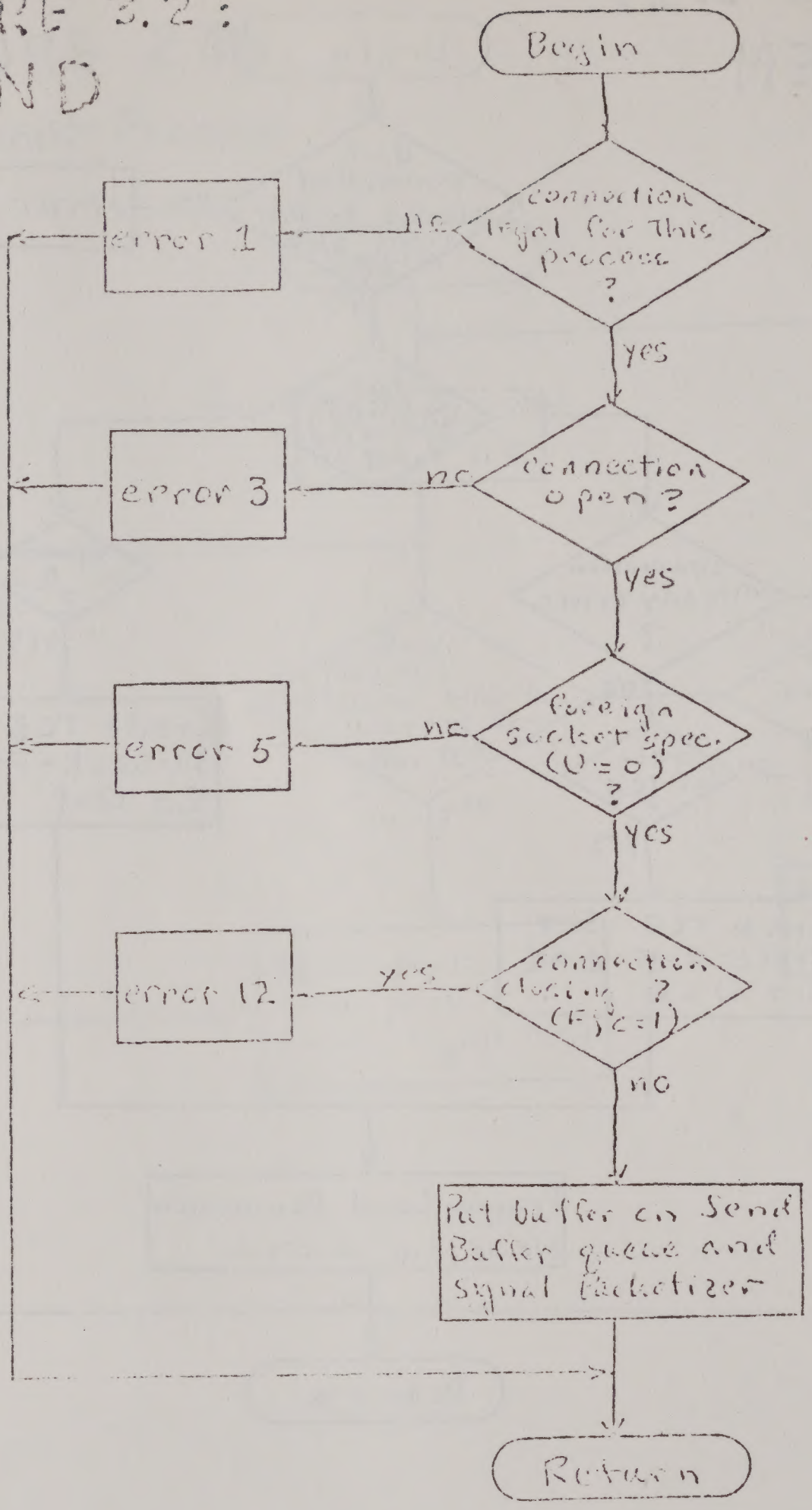


FIGURE 3.3:
INTERRUPT

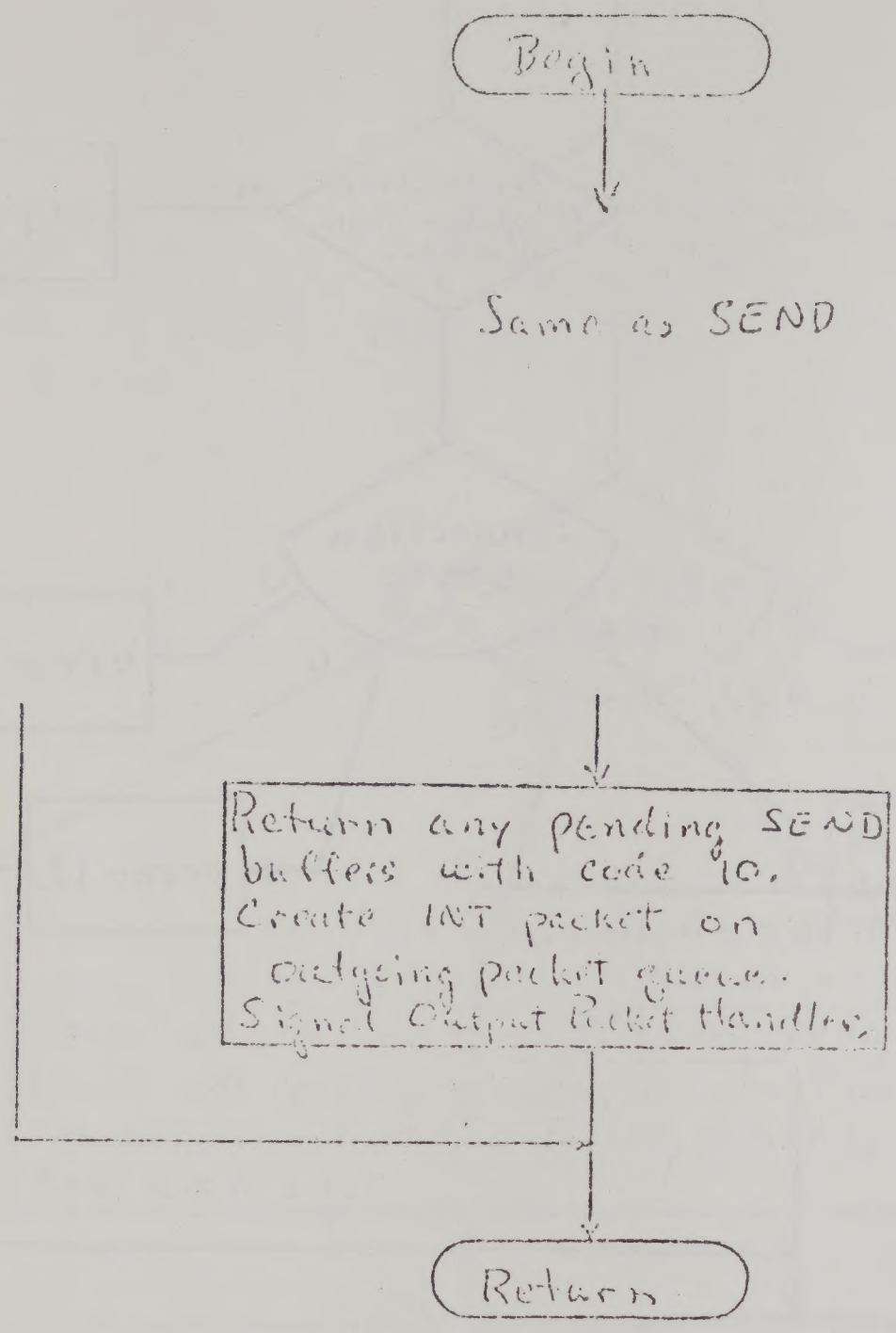

Begin
Same as SEND
Return any pending SEND buffers with code 10.
Create INT packet on outgoing packet queue.
Signal Output Packet Handler.
Return

FIGURE 3.1:
RECEIVE

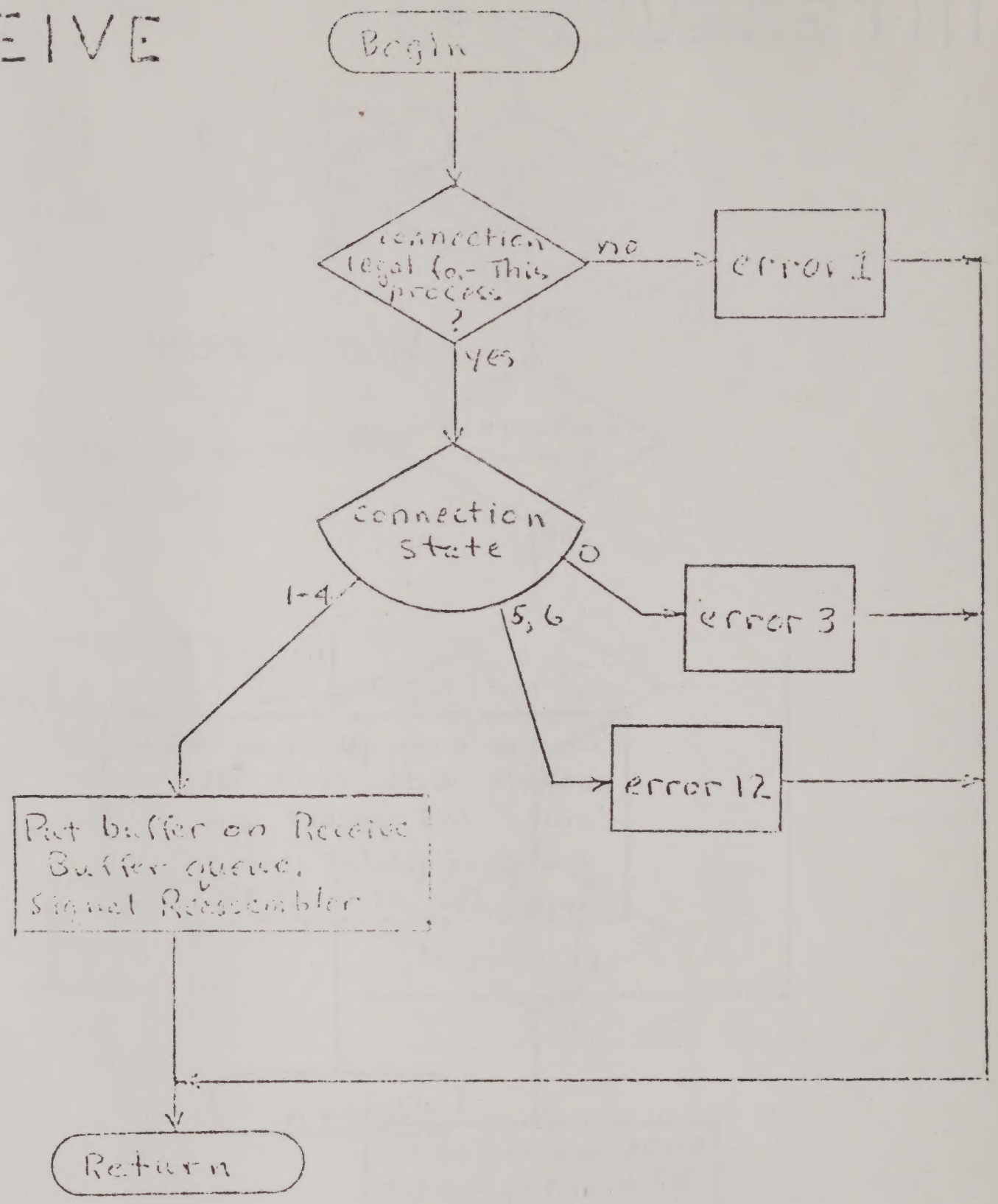


FIGURE 3.5:
CLOSE

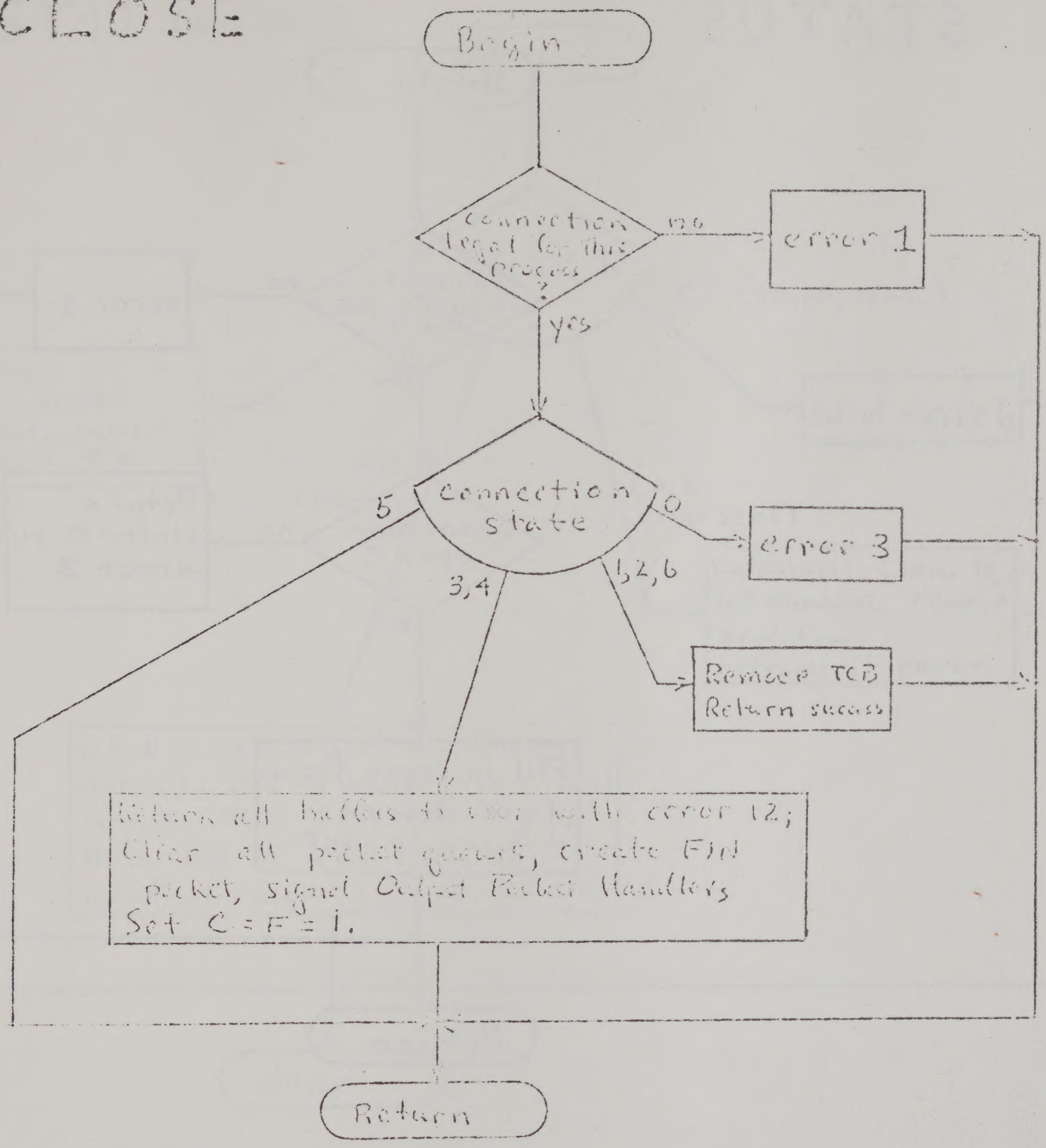


FIGURE 3.6 :
STATUS

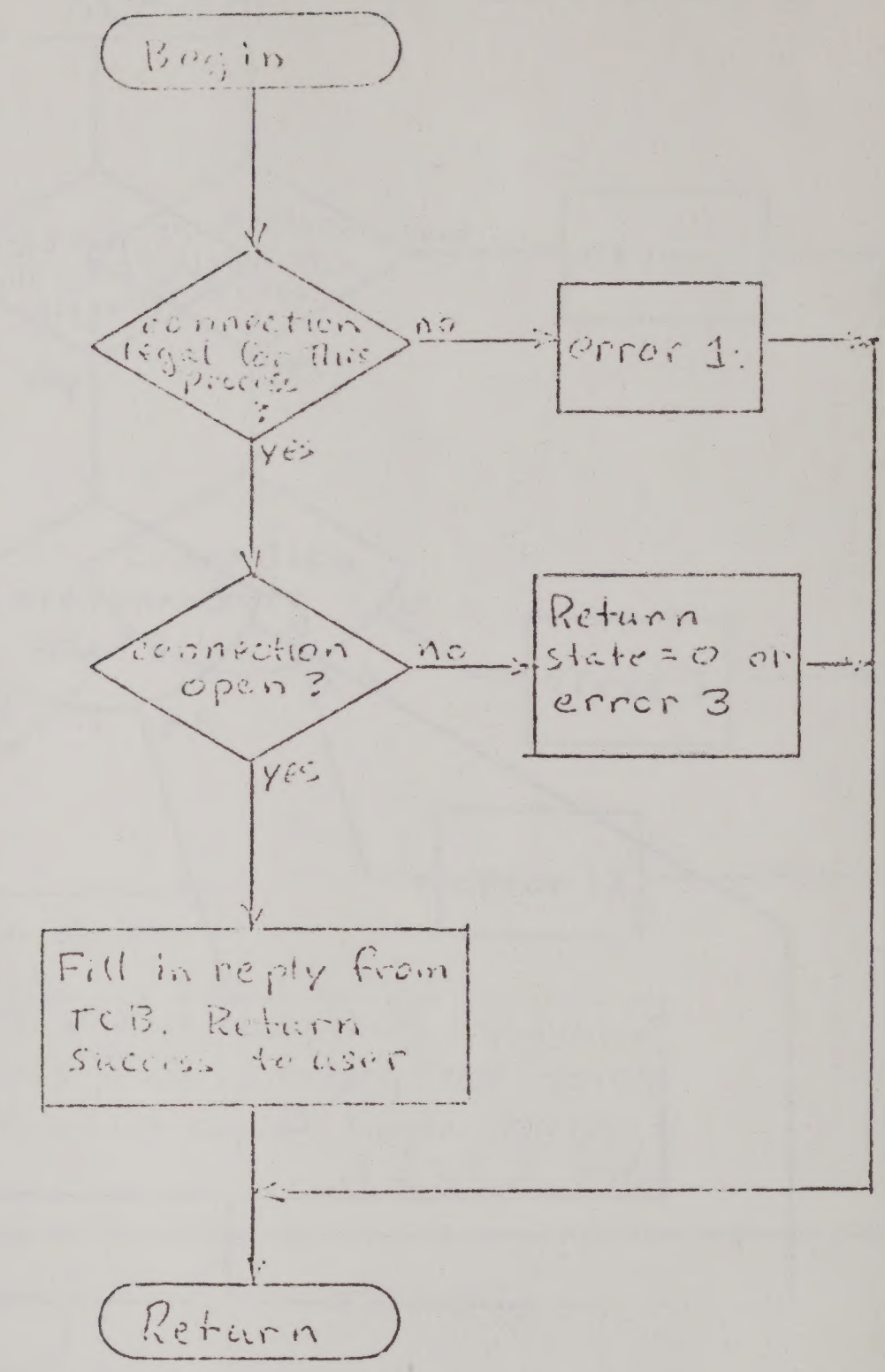


FIGURE 9.1 :
SYN (no ACK)

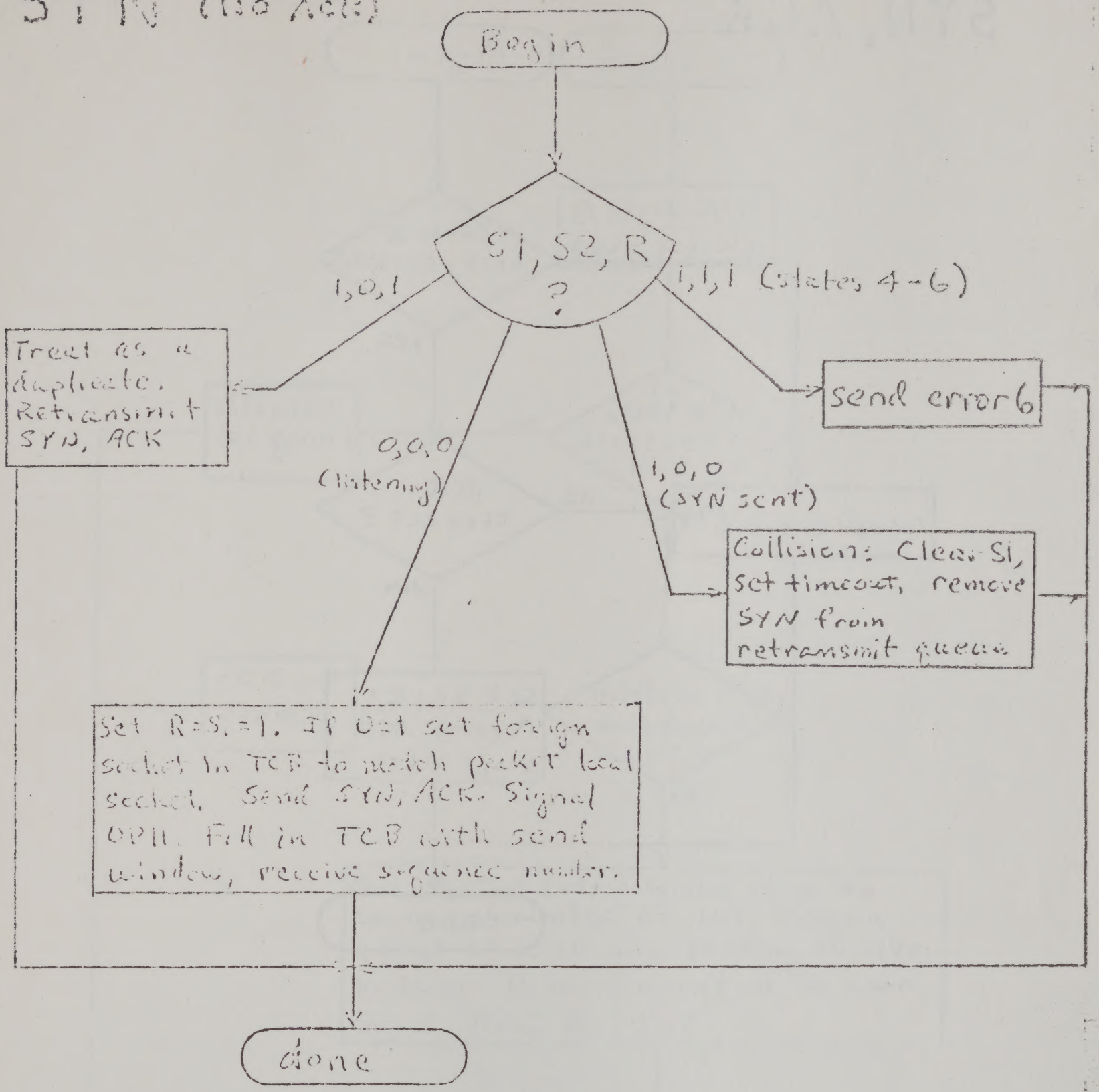

Begin
S1, S2, R
?
1,0,1
1,1,1 (states 4-6)
Treat as a duplicate. Retransmit SYN, ACK
send error 6
0,0,0
(listening)
1,0,0
(SYN sent)
Collision: Clear S1, set timeout, remove SYN from retransmit queue
Set R=S.=1. If O=1 set foreign socket in TCB to match packet local socket. Send SYN, ACK. Signal OPN. Fill in TCB with send window, receive sequence number.
done

FIGURE 4.2:
SYN, ACK

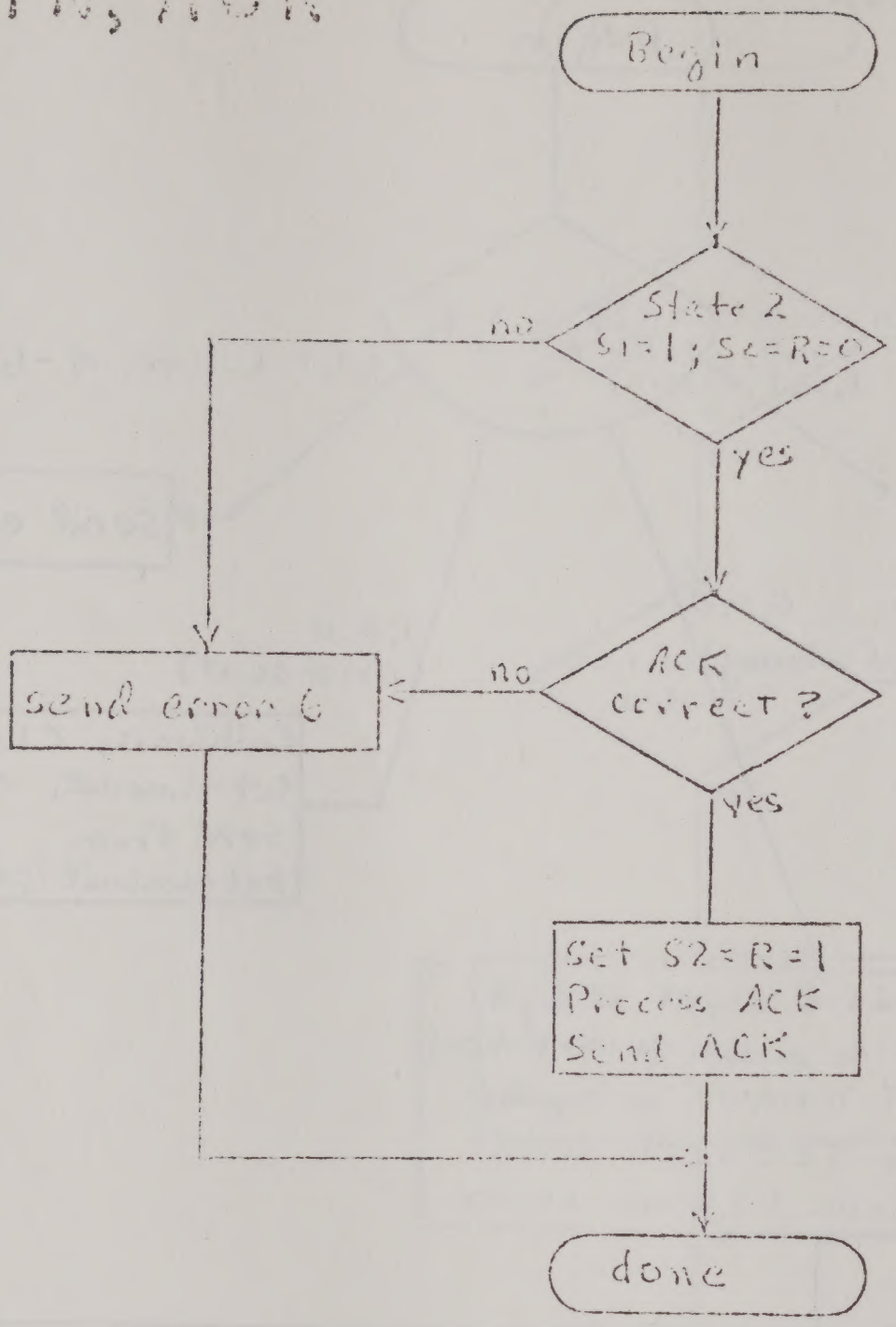


FIGURE 4.3:
INT (Connect)

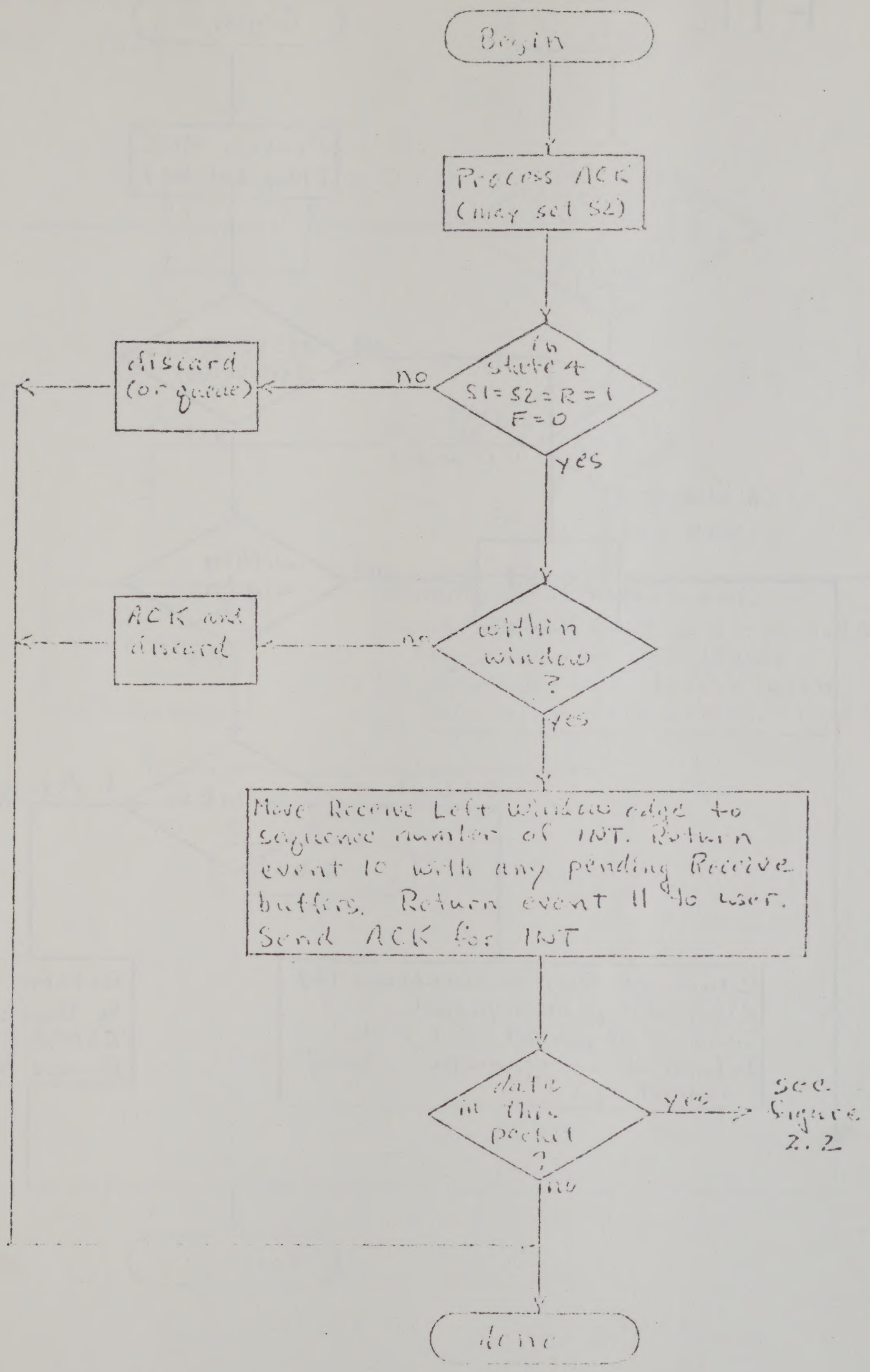


FIGURE 1.4:
FIN

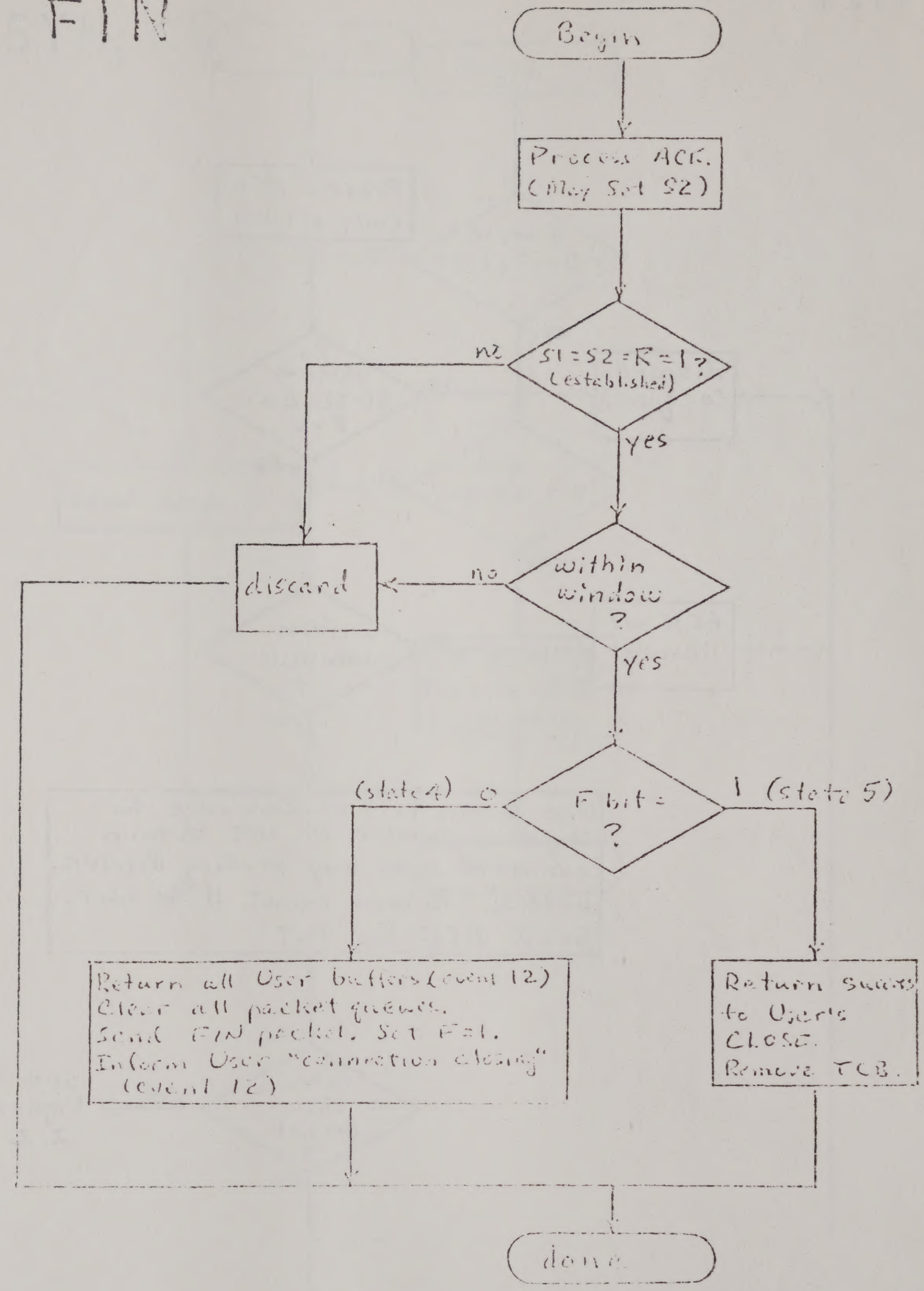
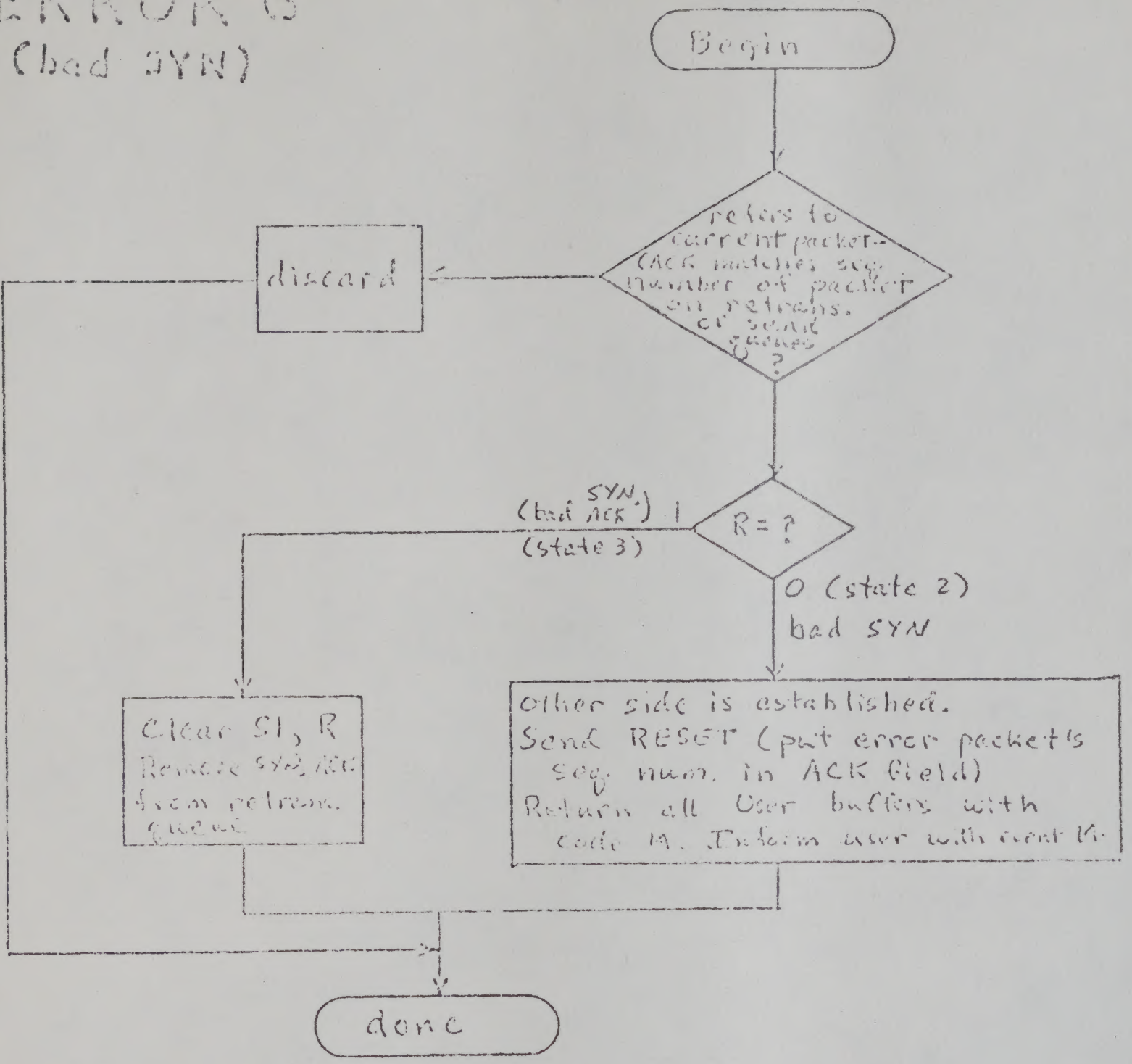


FIGURE 4.5:
ERROR 6
(bad SYN)


Begin
refers to current packet. (ACK matches seq. number of packet on retrans. or send queues ?
discard
R = ?
(bad SYN, bad ACK) 1 (state 3)
0 (state 2) bad SYN
Clear SI, R
Remove SYN, ACK from retrans. queue
other side is established.
Send RESET (put error packet's seq. num. in ACK field)
Return all User buffers with code 14. Inform user with next U-
done

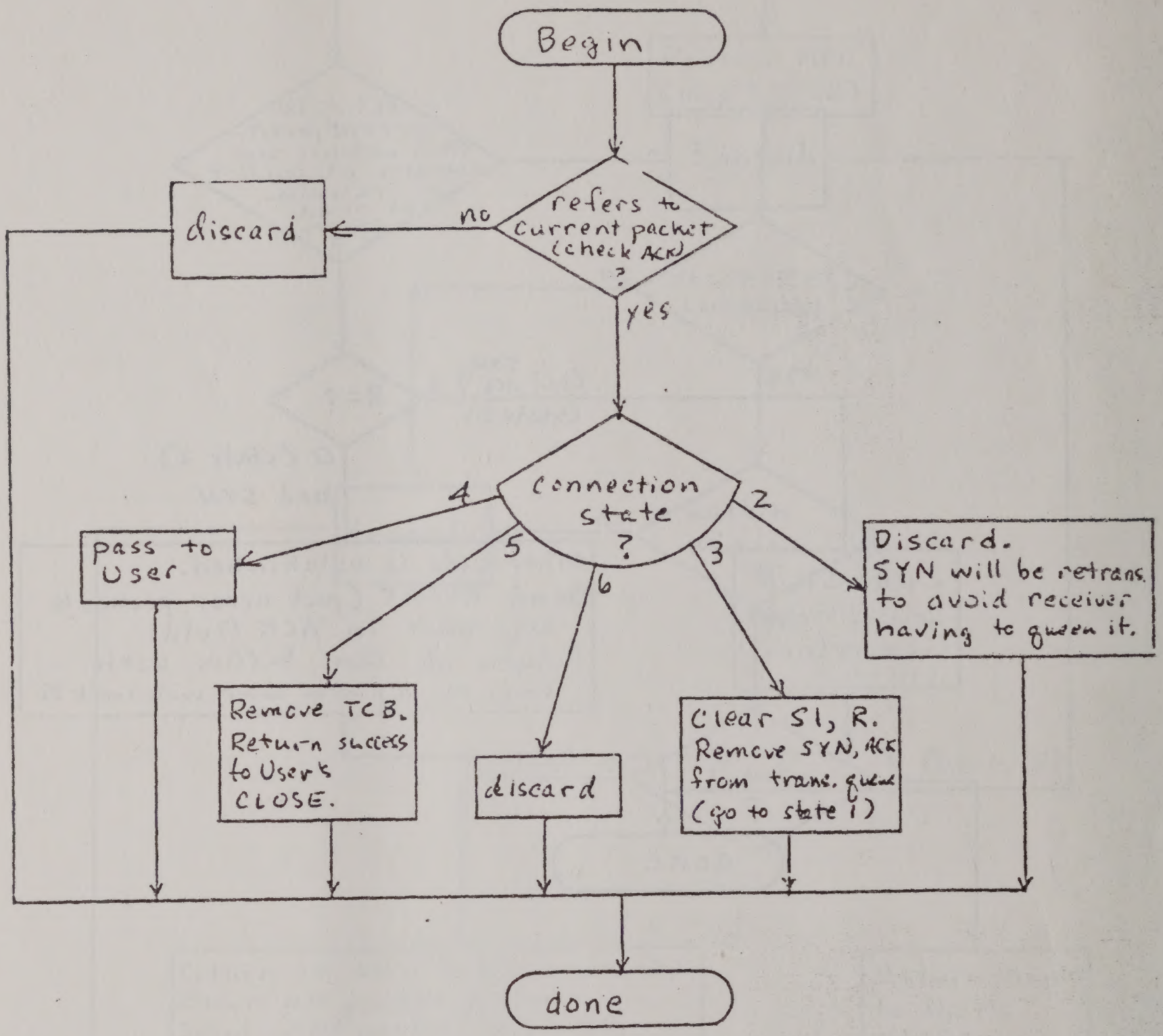



FIGURE 4.6:
ERROR 7,8